ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACIÓN

UNIVERSIDAD DE CANTABRIA

Trabajo Fin de Máster

Desarrollo de un entorno de visualización de datos de smart-city basados en NGSI

(Development of a smart-city data visualization environment based on NGSI)

Para acceder al Título de

Máster Universitario en Ingeniería de Telecomunicación

Autor: Mario Núñez Callejo

Junio - 2021

AGRADECIMIENTOS

Reservo este pequeño espacio para agradecer al director de este Trabajo de Fin de Máster, Luis Diez, tanto por brindarme la oportunidad de realizar este proyecto como por el tiempo dedicado y el apoyo proporcionado durante la realización del mismo.

Fuera del aspecto técnico también agradezco el apoyo recibido por parte de mi familia, la cual siempre ha confiado en mi y no ha permitido que flaquee en los momentos más difíciles.

RESUMEN

El desarrollo de las TIC ha supuesto un gran avance para multitud de sectores, lo cual ha generando un crecimiento en la calidad de vida de las personas a lo largo de los últimos años. Este desarrollo ha dado lugar a la creación de nuevos paradigmas, como es el caso de la Internet de las Cosas (IoT) el cual se basa en la interconexión de objetos físicos para obtener información del entorno.

Uno de los casos de uso más relevantes de la IoT es el de las ciudades inteligentes (*smart cities*), las cuales son localidades con un gran despliegue de tecnología IoT que permite, entre otros aspectos, la monitorización de diferentes parámetros. A su vez, esta información puede ser utilizada para el desarrollo de nuevos servicios, o para la optimización de otros existentes mediante su digitalización.

En el escenario actual, cada ciudad posee dispositivos IoT de fabricantes diferentes que exponen la información usando interfaces propietarios. Esto obliga a adaptar cada servicio urbano que se desarrolle al formato que siguen los datos en cada una de esas ciudades, lo que supone un coste elevado y plantea una solución ineficiente a la hora de reutilizar servicios ya generados en otros lugares.

Para solucionar estas deficiencias se evidencia la necesidad de uniformizar tanto la estructura general de los datos como cada uno de los campos que posee cada tipo de datos. El uso de estándares e iniciativas abiertas hace posible esta homogeneización en los datos, lo cual supone una gran ventaja a la hora de generar y replicar servicios, ya que podrán ser desplegados fácilmente en los diferentes entornos que sigan estos estándares.

Bajo esta premisa, en este proyecto se han desarrollado y validado servicios de procesado y representación de datos utilizados en la plataforma de *smart city* de la ciudad de Santander. Dichos servicios se basan en el estándar NGSI y la iniciativa global *Smart Data Models*, los cuales dotan a los datos de una consistencia que hace que el despliegue de dichos servicios en cualquier localidad que siga estos estándares no requiera de una adaptación específica. Además, con el fin de aportar servicios de valor añadido, se desarrolla un módulo de análisis de datos que realiza predicciones sobre el valor que van a tener determinados datos en el futuro.

Palabras Clave— IoT SmartSantander, NGSI, Smart Data Models, Elastic Stack.

ABSTRACT

The development of ICT has made a great impact in many sectors, which has generated a growth in the quality of life of people over the last few years. This development has lead to the creation of new paradigms, such as the Internet of Things (IoT) which is based on the interconexion of physical objects to obtain environmental information.

One of the most relevant use cases of IoT is the smart cities, which are urban ecosystems with a large deployment of IoT technology that allows, among other aspects, the monitoring of different parameters. In turn, this information can be used for the development of new services, or for the optimization of existing ones by digitizing them.

In the current scenario, each city has IoT devices from different manufacturers that expose the information using their own interfaces. This makes it necessary to adapt each urban service that is developed to the format followed by the data in each of these cities, which implies a high cost and poses an inefficient solution when it comes to reusing services already generated elsewhere.

To solve these deficiencies, it is required the standardization of both the general structure of the data and each of the fields that each type of data has. The use of open standards and initiatives makes this homogenization of data possible, which is a great advantage when generating and replicating services, since they can be easily deployed in environments that follow these standards.

Under this premise, this project has developed and validated data processing and representation services used in the smart city platform of the city of Santander. These services are based on the NGSI standard and the global initiative *Smart Data Models*, which provide the data with a consistency that makes the deployment of these services in any location that follows these standards does not require a specific adaptation. In addition, in order to provide value-added services, a data analysis module is developed which makes predictions about the value that certain data will have in the future.

Keywords— IoT SmartSantander, NGSI, Smart Data Models, Elastic Stack.

Índice

Índice de Figuras **III**

Índice de Tablas **IV**

Lista de Acrónimos **IV**

1 Introducción **1**

1.1 Motivación 2

1.2 Objetivos 3

1.3 Estructura del documento 4

2 Marco tecnológico **5**

2.1 IoT 5

2.2 Smart Cities 6

2.3 FIWARE 8

3 Herramientas y recursos utilizados **15**

3.1 Interfaz NGSI 15

3.2 Smart Data Models 23

3.2.1 Definición de modelos de datos 25

3.3 Elastic stack 27

3.3.1 Elasticsearch 28

3.3.2 Kibana 31

3.3.3 Logstash 35

3.3.4 Beats 41

4 Implementación **43**

4.1 Arquitectura del sistema 43

4.2 Obtención y almacenamiento de los datos 47

4.3 Representación de los datos 59

4.4 Técnicas de analítica de datos 73

5 Conclusiones y líneas futuras **81**

5.1 Conclusiones 81

5.2 Líneas futuras 82

Bibliografía **85**

Índice de Figuras

Capítulo 2

2.1 Infraestructura de SmartSantander 8
2.2 Componentes de FIWARE 10

Capítulo 3

3.1 Modelo de datos genérico 16
3.2 Representación entidad 18
3.3 Representación atributo 18
3.4 Representación de una localización de tipo `geo:point` 20
3.5 Representación de una localización de tipo GeoJSON 21
3.6 Flujo de datos Logstash, Elasticsearch y Kibana 36
3.7 Componentes de Logstash 36
3.8 Flujo de los datos del Elastic Stack completo 41

Capítulo 4

4.1 Arquitectura del sistema completo 44
4.2 Respuesta del *endpoint* donde se está ejecutando Elasticsearch 48
4.3 Estructura general de un archivo de configuración de *pipeline* de Logstash . . . 49
4.4 Etapa de *input* de Logstash `TrafficFlowObserved` 51
4.5 Etapa *input* de Logstash múltiples peticiones 51
4.6 Transformación en etapa *filter* de `TrafficFlowObserved` 53
4.7 Etapa *output* de Logstash de `TrafficFlowObserved` 55
4.8 Petición para modificar el tipo del campo `geolocation` 56
4.9 Mapeo de la localización en `OnStreetParking` 57
4.10 Petición para modificar el tipo de los campos de geolocalizaciones en `OnStreetParking` 58
4.11 Serie temporal que representa la variación de la intensidad de tráfico 60

4.12 Histograma con los valores medios de intensidad de varios sensores de tráfico . 61
4.13 Mapa con las geolocalizaciones de los sensores de tráfico 62
4.14 *Tooltip* con imagen en el mapa generado para `PointOfInterest` 63
4.15 Mapa del *Smart Data Models* `OnStreetParking` 64
4.16 Tabla informativa de los datos del *Smart Data Model* `ForecastWeather` . . . 65
4.17 *Dashboard* del *Smart Data Model* `PointOfInterest` sin filtrar 67
4.18 *Dashboard* de los valores de intensidad de tres sensores del *Smart Data Model* `TrafficFlowObserved` 68
4.19 Página de inicio con enlaces a los *dashboards* generados 69
4.20 *Random Forest* 76
4.21 *Grid Random Forest* 76
4.22 *Neural Network* 77
4.23 *Linear Regression* 77
4.24 *Dashboard* con series temporales de intensidad de tráfico media real y predicha 79

Índice de Tablas

Capítulo 4

4.1 Análisis de *Smart Data Models* 72

Lista de Acrónimos

TIC Tecnologías de la Información y la Comunicación

IoT Internet of Things

JSON JavaScript Object Notation

NGSI Next Generation Service Interface

ML Machine Learning

SSO Single Sign-On

PEP Performance-Enhancing Proxies

PAP Policy Administration Points

PDP Policy Decision Points

GSMA Global System for Mobile Communications

CEF Connecting Europe Facility

ETSI European Telecommunications Standards Institute

XACML eXtensible Access Control Markup Language

MIME Multipurpose Internet Mail Extensions

WGS World Geodetic System

URL Uniform Resource Locator

ELK Elasticsearch Logstash Kibana

HTTP Hypertext Transfer Protocol

MQTT Message Queuing Telemetry Transport

M2M Machine To Machine

API Application Programming Interface

REST REpresentational State Transfer

CRUD Create Read Update Delete

ROS Robot Operating System

JVM Java Virtual Machine

SVM Support Vector Machine

SVR Support Vector Regression

Capítulo 1

Introducción

En los últimos años se ha asistido a un continuo avance en las tecnologías de la información y comunicación (TIC). Este avance ha tenido tanto un componente técnico como de asimilación por parte de la sociedad. Esto ha llevado tanto a la aparición de nuevos servicios, como a la adaptación y optimización de otros existentes.

Un claro ejemplo de esto es el despliegue de dispositivos IoT en las ciudades, lo que ha dado lugar al término *smart city*. En estos contextos urbanos la adopción de soluciones TIC permite recopilar información desde diversas fuentes y, de este modo, mejorar la toma de decisiones de servicios. Esto redunda en una mejora de la gestión de las ciudades y de la provisión de servicios a la ciudadanía.

Ante este contexto, en este proyecto se va a realizar un despliegue de servicios en la *smart city* de Santander que permitan la representación uniforme de datos de contexto de la ciudad. La generación de estos servicios ha sido realizada de tal manera que, a pesar de haber sido desarrollados para una ciudad en concreto, poseen un carácter flexible lo que permite que puedan ser fácilmente extendidos a otras localidades sin necesidad de realizar complejas configuraciones.

En las siguientes secciones de este capítulo se da una visión general de cual ha sido la motivación del proyecto. También se proporciona la estructura principal del documento y una breve explicación del contenido de cada capítulo.

1.1. Motivación

Las grandes cantidades de datos generadas en las *smart cities* gracias a la IoT son una importante fuente de información que puede utilizarse para generar servicios y aportar un mayor nivel de calidad de vida a los habitantes de dichas ciudades [1].

El disponer de datos abiertos y accesibles, así como de sistemas abiertos, es un aspecto clave para que las industrias y la ciudadanía puedan desarrollar nuevos servicios y aplicaciones. Es por esto que se debe tener en cuenta que la utilidad de los datos está sujeta a que existan servicios que puedan hacer uso de ellos y mostrarlos de la forma más apropiada para los usuarios [2].

En cada *smart city* los dispositivos que captan la información pueden pertenecer a diferentes fabricantes, por tanto el formato de sus datos también puede ser diferente. Es tarea de los servicios acceder a esos datos, con formatos heterogéneos, y utilizarlos de la manera más conveniente para la ciudad. Esta situación, a pesar de que permite generar servicios que hagan uso de los datos almacenados, es ineficiente en términos de extensibilidad.

La ineficiencia viene dada debido a que si se quiere desplegar un determinado servicio en muchas ciudades y cada una de ellas utiliza un formato distinto de datos, supondría tener que adaptar ese servicio a todas ellas. La adaptación de servicios a los formatos definidos por cada una de las distintas entidades supone un elevado coste y un desperdicio de recursos considerable. Por otro lado, el despliegue de nuevos servicios puede verse condicionado por los existentes, creando una dependencia de los entornos TIC urbanos con un determinado fabricante o proveedor de servicio, lo que se ha venido a llamar *vendor lock-in*.

Ante este escenario, han aparecido diferentes iniciativas que buscan fomentar la interoperabilidad de servicios. Esto afecta tanto a los interfaces, como a los modelos de información y datos. Los primeros hacen referencia a la estructura y relación de las entidades de información, mientras que los segundos son específicos de cada tipo de dato y definen qué campos poseen. Uno de los interfaces más extendidos es el *Next Generation Service Interface* (NGSI), que provee tanto el formato general de los datos como los métodos para acceder a ellos. En base a él, en los últimos años ha aparecido una iniciativa de modelo de datos, *Smart Data Model*, que busca un consenso en la representación de los mismos.

Aunque la definición de estándares es un requisito para posibilitar la interoperabilidad por sí solo puede no ser suficiente. Hoy en día existen entornos y plataformas (*frameworks*) de desarrollo ampliamente utilizados por la comunidad de desarrolladores. Esto lleva a que tengan un mejor soporte, documentación y, en general, aceptación. Lo que a su vez permite que existan herramientas de desarrollo que se hayan convertido en estándar *de-facto* y es en estas en las que los nuevos estándares se tienen que integrar.

1.2. Objetivos

Una vez expuesta la motivación del proyecto, se determina el objetivo principal de este Trabajo de Fin de Máster como el desarrollo de un sistema de visualización de la información basado en modelos de información y datos estándares, implementado sobre uno de los entornos de representación de información más ampliamente utilizados, Elastic Stack. Asimismo, se han desarrollado las funcionalidades básicas que permitan utilizar técnicas de *Machine Learning* (ML) sobre los datos, y su integración en el entorno de visualización a fin de proporcionar información de valor añadido.

En concreto, este trabajo ha tenido los siguientes objetivos:

- Conexión y extracción de datos del API estándar NGSI donde está almacenada toda la información recogida por los dispositivos IoT desplegados por la ciudad.
- Transformación consistente de los datos que siguen los modelos estándares y almacenamiento de dicha información en una base de datos específica para funciones de búsqueda.
- Representación uniforme de los datos almacenados para la generación de visualizaciones gráficas que permitan un consumo sencillo y visual de la información por parte de los usuarios.
- Desarrollo de servicios de valor añadido mediante la generación de un módulo de análisis de datos que permita realizar predicciones de valores futuros en base a los datos ya almacenados.

1.3. Estructura del documento

La estructura del presente documento está basada en cinco capítulos principales, los cuales conforman una guía del trabajo realizado a lo largo de este proyecto y de los elementos que forman parte del mismo. Precediendo a estos capítulos principales se encuentra definido un resumen y un *abstract*, cuyo objetivo es el de aportar una visión general de todo el proyecto recogiendo los aspectos más importantes del mismo.

El primero de los capítulos es introductorio, en él se presentan las circunstancias que han motivado al desarrollo del proyecto y los objetivos que se pretenden alcanzar.

En el segundo capítulo se analiza el marco tecnológico en el que se engloba este proyecto. Dentro de este contexto se hace una revisión de las tecnologías y entidades sobre las que se ha sustentado el trabajo realizado.

Seguidamente, el tercer capítulo se centra en explicar en profundidad las herramientas y recursos que se han utilizado y que han sido un pilar fundamental para poder llevar a cabo la implementación del proyecto.

Una vez introducidos los conceptos básicos y las herramientas que van a utilizarse, el cuarto capítulo detalla de manera exhaustiva el sistema implementado y el procedimiento seguido para lograr dicha implementación. Además, también se describen los elementos de dicho sistema que no han sido desarrollados, pero que ha sido necesaria una interacción con ellos para lograr la funcionalidad final.

Después de haber descrito la arquitectura global del sistema con cada uno de los elementos que la componen, se finaliza el documento analizando las conclusiones a las que se ha llegado y se exponen una serie de ideas, las cuales podrían servir como punto de partida para futuras ampliaciones de este trabajo.

Capítulo 2

Marco tecnológico

El presente trabajo de fin de máster se sustenta en una serie de iniciativas que previamente se fomentaron y ganaron relevancia. En este capítulo se realiza una revisión del marco tecnológico con lo que se busca contextualizar el proyecto desarrollado. En primer lugar se presentará el concepto de Internet de las Cosas, para seguidamente presentar su aplicación a los entornos urbanos, dando lugar a las ciudades inteligentes. A continuación se presentará la iniciativa FIWARE, sobre la que se articula este trabajo.

2.1. IoT

El concepto de IoT (por sus siglas en ingles, *Internet of Things*), también conocido en el habla hispana como Internet de las Cosas, fue utilizado por primera vez en 1999 [3] y desde entonces ha ido cobrando cada vez más relevancia a lo largo de los años.

La IoT cambió el concepto que se tenía sobre las redes e Internet para la mayoría de usuarios. Antes de su aparición los contenidos y servicios a los que se podía acceder a través de Internet tenían mayormente un origen humano, es decir, eran personas las que generaban datos y hacían efectiva su disponibilidad para otros usuarios.

Este modelo basado en que las personas generen los datos es, en ciertos casos, inefectivo debido a que las limitaciones que tienen los humanos a la hora de generar y hacer disponibles ciertos datos se extrapolan al servicio que se está dando. Es decir, que el servicio queda limitado

por la capacidad humana.

Uno de los escenarios que ejemplifican mejor esta limitación es en el caso de extraer datos del entorno. Previo a la IoT, si se querían hacer disponibles datos del entorno en Internet se tenían que medir, almacenar y cargar en un servidor. Este proceso era realizado por máquinas controladas por personas, lo cual lo hacía lento y estaba sujeto a posibles fallos de las propias personas, algo intrínseco a la naturaleza humana.

En cambio, con la IoT se disponen de dispositivos electrónicos capaces de medir magnitudes del entorno y enviarlas a un servidor de manera automática, todo esto sin interacción humana. La información del ambiente se recoge por unos elementos llamados sensores, los cuales están conectados entre sí y enviarán su información medida a otro dispositivo que concentra los datos generados. Este aparato que recibe los datos se llama *gateway*. y su función es enviar la información fuera de la red de sensores, a un servidor por ejemplo. De este modo, y de manera automática, los datos ya estarían disponibles en la red para ser consumidos por los usuarios que lo deseen o que estén autorizados a hacerlo.

2.2. Smart Cities

El avance de la tecnología electrónica, así como de la IoT, ha permitido que surjan nuevos modelos de negocio y que muchas compañías decidan ofrecer servicios basados en esta tecnología. Uno de los conceptos que ha generado la IoT es el de Smart City (ciudad inteligente), el cual se ha expandido por todo el mundo, comenzando así, un desarrollo tecnológico a nivel metropolitano con grandes expectativas de futuro.

La tendencia gregaria de los seres humanos generó, hace ya muchos años, la formación de grupos sociales en torno a ciertos lugares, lo que se conoce como ciudades. Uno de los afanes de estos grupos de personas localizados, ha sido siempre la mejora de la calidad de vida de los integrantes de la comunidad. Una de las formas más comunes de conseguir esto es la construcción de un hábitat más confortable, lo cual, comenzó en sus inicios con la construcción de viviendas, generación de comercios, y más adelante siguió con la creación de infraestructura de carreteras, parques y zonas verdes, entre otras cosas.

En la actualidad, se sigue el mismo principio que cuando se formaron las ciudades, solo que en este caso se busca aportar mayor valor a la ciudad mediante mejoras tecnológicas. La idea que subyace a todo ese desarrollo tecnológico es la de poder ofrecer a los ciudadanos una gran variedad de servicios, así como obtener información del entorno para poder tomar decisiones y mejorar la eficiencia, a nivel de recursos, de la ciudad. Esto es lo que se conoce como ciudad inteligente.

La mejora en la gestión de los recursos de la ciudad tiene como principal beneficio un ahorro económico, además de mayor control sobre el estado en el que se encuentra la comunidad a nivel de infraestructura y también ambiental. Esto sumado a que el despliegue de todos estos servicios resulta muy atractivo, no solo para los ciudadanos, sino también para potenciales visitantes que quieran disfrutar de ellos, es lo que ha hecho que muchas de las ciudades más importantes del mundo decidan aplicar este modelo.

Entre las ciudades inteligentes más desarrolladas del mundo se pueden encontrar nombres como Dubai, la cual ha planificado digitalizar todos los servicios gubernamentales, además de cientos de iniciativas relacionadas con el transporte, comunicaciones, infraestructura y electricidad. También se conoce que ciudades como Nueva York, Londres y Hong Kong, entre otros, poseen grandes redes de sensores desplegadas para captar información y aportar servicios, lo que ha llevado a posicionarlas como unas de las ciudades inteligentes más grandes del planeta. Otras localidades centran su avance tecnológico para combatir el cambio climático y enfocan la ciudad inteligente como una oportunidad para favorecer la sostenibilidad del medio ambiente, es el caso de Oslo. La capital noruega cuenta con una gran variedad de sensores para controlar la iluminación y los niveles de gases en el ambiente entre otras cosas, cuyos datos servirán para reducir las emisiones de sustancias nocivas para el medio ambiente [4].

El presente trabajo de fin de máster va a desarrollarse entorno a los servicios aportados por la plataforma SmartSantander, cuyo desarrollo permitió incorporar soluciones de *Smart City* a la ciudad de Santander. SmartSantander nace como un proyecto europeo de investigación [5] que desarrolla una infraestructura para el desarrollo de arquitecturas, tecnologías facilitadoras esenciales, servicios y aplicaciones para la Internet de las Cosas. El proyecto SmartSantander supone el despliegue de miles de sensores por toda la ciudad y la puesta en marcha de varias aplicaciones que utilizan la información que recogen para ponerla a disposición de los ciudadanos.

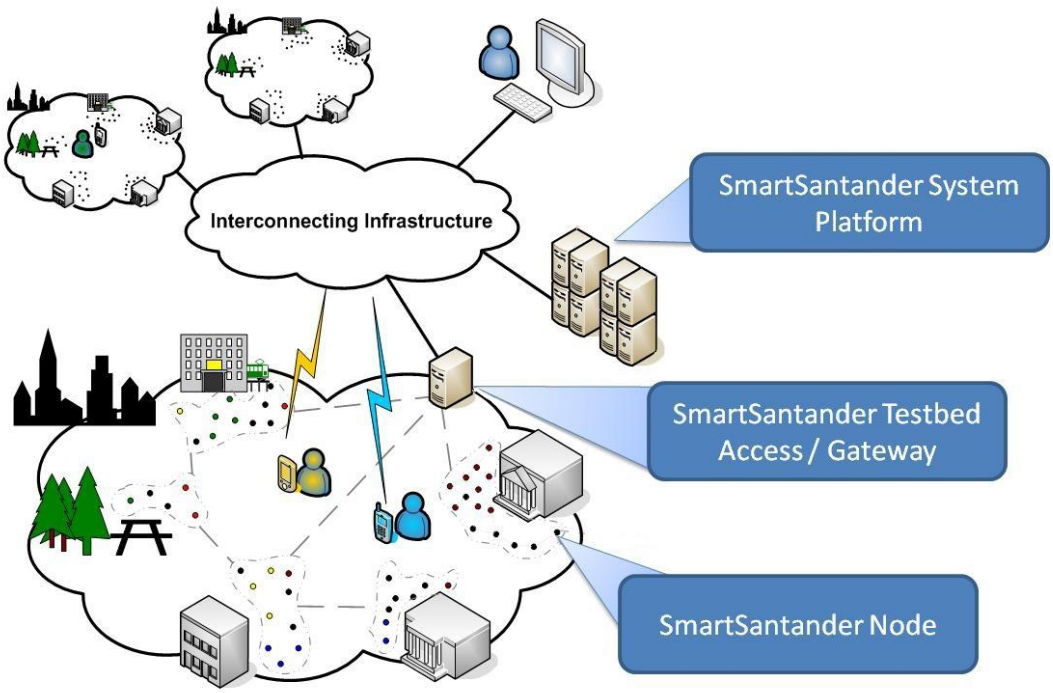

Figura 2.1: Infraestructura de SmartSantander

Tal como se observa en la arquitectura global de SmartSantander en la Figura 2.1, los sensores captan información del entorno, después esa información es enviada a un *gateway* que es quien finalmente envía a los servidores de SmartSantander los datos recibidos. Ahí se almacenan las observaciones captadas por los sensores a la espera de ponerse a disposición de los clientes que deseen beneficiarse de dicha información.

Entre algunos de los servicios que aporta SmartSantander, y de los cuales se hablarán en los siguientes capítulos del presente documento, se encuentran algunos como conocer los aparcamientos disponibles, los niveles de contaminación, temperatura en diferentes zonas y previsión meteorológica, entre otros.

2.3. FIWARE

Para lograr aprovechar de una manera más eficiente la tecnología de la que se está haciendo uso hoy en día, aparecen organizaciones como FIWARE, cuya misión principal se centra en fomentar una serie de estándares para un mejor aprovechamiento de los datos en las conocidas

como Smart Solutions, entre las que se encuentran las Smart Cities. Es por esto por lo que FIWARE cobra especial importancia en estos sistemas formados por dispositivos inteligentes y altamente interconectados que tienen como objetivo proveer servicios con mejores prestaciones.

Para lograr comprender la labor que desarrolla FIWARE, se ha de volver al plano físico en el que se encuentran los dispositivos IoT. El objetivo principal de la Internet de las Cosas, así como de otras tecnologías, es lograr obtener información para, de un modo u otro, generar una mayor confortabilidad en la vida de las personas. Esta idea se sustenta principalmente en el concepto de la información de contexto (*context data*), los cuales se pueden definir como todos aquellos datos que de algún modo tiene relación con lo que está sucediendo en un determinado ámbito.

Un claro ejemplo de este concepto se encuentra en el ámbito de las ciudades inteligentes. Una localidad cualquiera tiene a su alrededor mucha información que puede ser de interés conocer para brindar servicios a los ciudadanos. Esos datos podrían obtenerse mediante diferentes procedimientos y acabar conociendo información como la temperatura, la velocidad del viento, los niveles de contaminación y la previsión meteorológica, entre otros. A todos estos parámetros, que de algún modo están relacionados con lo que está sucediendo en la ciudad y que podrían ser de utilidad para conocer más acerca de ella, se les conoce como datos de contexto.

La información que se relaciona con un cierto ámbito puede ser extraída de diferentes fuentes. La más habitual es la extracción de datos del entorno mediante el despliegue de dispositivos IoT cuya principal misión es la de realizar mediciones periódicas de magnitudes ambientales. Pero además existen otras fuentes de las que se puede extraer información, como son por ejemplo los servicios web que recogen datos sobre parámetros que aportan información acerca de la ciudad pero no han sido obtenidos mediante sensores, como es el caso de la previsión meteorológica.

Al recopilarse información de diferentes fuentes, es común la generación de silos, es decir, conjuntos de datos pertenecientes a ciertas fuentes que se mantienen aislados de las demás. Deshacerse de estos límites entre dominios permitirá el intercambio de datos través de múltiples aplicaciones, evitando así dichos silos de información. FIWARE está impulsando los estándares para hacer esto posible, liberando el potencial de los datos abiertos en tiempo real a gran escala. Esto simplifica la el desarrollo de servicios complejos, permitiendo a los desarrolladores unir información proveniente de la IoT con la de otras fuentes relevantes y poder acceder a ella usando

APIs REST bien definidos, lo que reduciría tiempo y coste de desarrollo. Los modelos de datos coordinados y los APIs abiertos permiten a los desarrolladores crear aplicaciones interoperables proporcionando así una mejor experiencia de usuario.

FIWARE es una organización que proporciona un *framework* basado en componentes de código abierto, los cuales pueden ser ensamblados junto con otros componentes de plataformas de terceros para acelerar el desarrollo de Smart Solutions. La arquitectura de FIWARE presenta cinco componentes principales, los cuales quedan representados en la siguiente imagen:

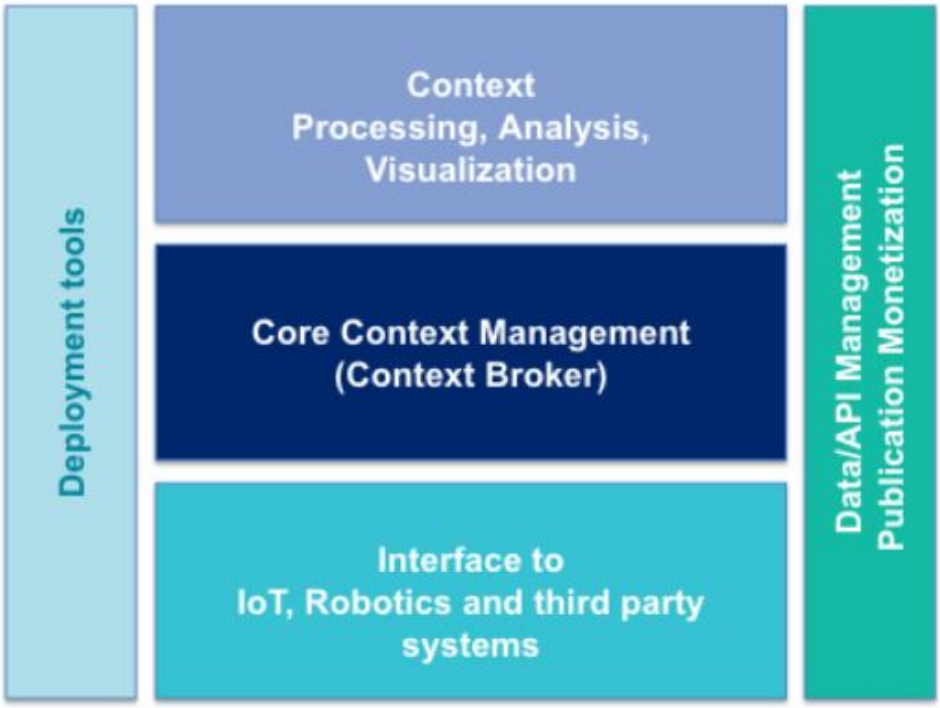

Figura 2.2: Componentes de FIWARE

En la Figura 2.2 se aprecia como todo gira entorno a la gestión de los datos de contexto, la cual se realiza por medio del *Context Broker*, el cual es el componente principal y el único obligatorio en cualquier sistema basado en FIWARE. El *Context Broker* es el componente que permite recibir información de contexto de diferentes fuentes, manteniendo su estado, y hacer accesible dichos datos a otros componentes, ya sean de FIWARE o de terceros, para su análisis, procesado y representación. Dicho componente proporciona una API que sigue el estándar NGSI, esto es una API REST simple pero versátil que permite realizar actualizaciones, consultas o suscribirse a cambios en la información del contexto. Esto proporciona un entorno

en el que se consigue una abstracción de la capa física, lo que permite la utilización de los datos sin importar cual sea el fabricante de los dispositivos así como los protocolos IoT que subyacen a ese API.

Inicialmente se adoptó el estándar OMA NGSI, para posteriormente modificarlo en base a la experiencia de uso dando lugar al llamado NGSIv2. En la actualidad se está migrando la implementación para dar soporte al estándar ETSI NGSI-LD, que extiende la funcionalidad de las versiones anteriores con la capacidad de enlazar datos de contexto.

Alrededor del *Context Broker*, FIWARE ofrece distintos componentes para complementar las funcionalidades proporcionadas, tal como se puede observar en la Figura 2.2. A continuación se describen brevemente estos componentes, conocidos también con el nombre de *Generic Enablers*, así como alguna de sus implementaciones más utilizadas [6]:

- ***Core Context Management***

  Este componente, el cual ya ha sido introducido con anterioridad, es la parte fundamental de la arquitectura de FIWARE y tiene como misión principal la manipulación y almacenamiento de la información de contexto para que pueda ser utilizada en posteriores procesos. Esta gestión de los datos la realiza de una manera altamente descentralizada y a gran escala.

  La implementaciones disponibles del *Context Broker* son las siguientes:

  - ***Orion Context Broker***: Es el *Context Broker* que está actualmente implementado y que proporciona la API FIWARE NGSI v2.
  - ***Orion-LD Context Broker***: Es un *broker* NGSI-LD que admite las APIs NGSI-LD y NGSI-v2. La versión beta actual está casi completa, y en un futuro se planea fusionarla con la rama principal de Orion.
  - ***Scorpio and Stellio Context Brokers***: son otros *brokers* NGSI-LD alternativos.

  Acompañando al *Context Broker* como parte del *Core Context Management* se encuentran ciertos *Generic Enablers*, de los cuales los más importantes se muestran a continuación:

  - ***Cygnus***: Se encarga de conservar las fuentes de información de contexto en otras bases de datos y sistemas de almacenamiento de terceros, creando una vista histórica del contexto.

- ***Draco***: Es un mecanismo alternativo de persistencia de datos para administrar el historial del contexto.
- ***Cosmos***: Permite un análisis de Big Data más simple en el contexto integrado con plataformas de Big Data (Spark y Flink).

- **Interfaz hacia los dispositivos IoT, robots o sistemas de terceros**

  Este componente tiene como principal objetivo el recibir la información de contexto de las diversas fuentes que la proveen, para después enviársela al *Context Broker* en el formato estándar NGSI. Esta interfaz hacia los dispositivos se compone de dos puertos: *North port* para la comunicación con el *Context Broker* y *South port* para escuchar el tráfico que mandan los dispositivos IoT.

  Gracias a esta capa se consigue independencia de la capa física, es decir, de los dispositivos y protocolos IoT, debido a que traduce el formato de los datos a NGSI, ya que es el único que admite el *Context Broker*. Las principales implementaciones de este componente son las siguientes:

  - **IDAS**: Proporciona una amplia variedad de *IoT Agents*, los cuales proporcionan una traducción entre diferentes protocolos de comunicación y el estándar NGSI. Los *IoT Agents* más importantes que proporciona IDAS son para los protocolos HTTP/MQTT con *payload* en formato JSON, Lightweight M2M, Ultralight y LoRaWAN entre otros. Además, proporciona una librería por medio de la cual es posible generar un *IoT Agent* personalizado según se necesite.
  - **FIROS**: Funciona como un traductor entre el dominio de la robótica y la nube, transformando los mensajes ROS (Robot Operating System) en NGSI y viceversa.
  - **Domibus**: Ayuda a los usuarios a intercambiar datos y documentos electrónicos entre sí de forma fiable.

- **Gestión del API, publicación y monetización**

  Este componente está dividido en dos partes bien diferenciadas. Una de ellas es la que permite controlar el acceso seguro y la gestión de los APIS, mientras que la otra se encarga de la publicación y la monetización de la información de contexto.

  La parte encargada de la seguridad y la gestión ofrece servicios y herramientas que permiten administrar la autenticación y autorización en aplicaciones y servicios *backend*,

proporcionando así un acceso seguro a los APIs. Si se desea administrar la identidad en aplicaciones sin necesidad de desarrollar los mecanismos oportunos, se puede ofrecer a los usuarios la posibilidad de iniciar sesión en la aplicación utilizando sus cuentas FIWARE. Las implementaciones que hacen posible el despliegue de estas funcionalidades son la siguientes:

  - **Keyrock Identity Management**: Es el encargado de la gestión de la identidad. Brinda soporte para la autenticación segura y privada de usuarios y dispositivos basada en el protocolo OAuth2, administración de perfiles de usuario, disposición de datos personales para preservar la privacidad, inicio de sesión único (SSO) y federación de identidades en múltiples dominios de administración.
  - **Wilma PEP Proxy**: Proporciona soporte para funciones de *proxy* dentro de esquemas de autenticación basados en OAuth2. También implementa funciones PEP dentro de un esquema de control de acceso basado en XACML.
  - **AuthZForce PDP/PAP**: Da soporte a las funciones PDP y PAP (administración y decisión de políticas) dentro de un esquema de control de acceso basado en el estándar XACML.
  - **APInf**: Es una herramienta para la administración de APIs. Proporciona todas las funciones necesarias para ejecutar negocios con API y facilita a los consumidores encontrar y comenzar a utilizar las API estándar. Sirve como un orquestador de Smart Cities que puede ser utilizado junto con otros componentes de FIWARE.

Por otro lado está la parte de publicación y monetización de la información de contexto. Debido a que dichos datos pasa por diferentes etapas, es posible comercializarlos en cualquiera de ellas, de modo que su valor será distinto dependiendo del grado de procesamiento que tengan.

- **Procesamiento y visualización de datos de contexto**

  Tiene como misión recibir los datos del *Context Broker* y realizar análisis, procesado y representación de la información de contexto. Algunos de los componentes más importantes que ofrece FIWARE para lograr este propósito son los siguientes:

  - **Wirecloud**: Ofrece una potente plataforma de *mashup web* que facilita el desarrollo de paneles operativos que son altamente personalizables por los usuarios finales.
  - **Kurento**: Permite el procesamiento en tiempo real de transmisiones multimedia.

- **FogFlow**: Soporta el procesamiento dinámico en la nube.
- **Perseo**: Introduce el procesamiento en tiempo real de eventos de contexto. Esto lo consigue utilizando un sistema basado en reglas, lo que le permite disparar eventos que envían solicitudes HTTP, correos electrónicos, *tweets*, mensajes SMS, etc.

Todo el sistema de FIWARE está orientado hacia la gestión de la información de contexto, lo que da lugar a que el *Context Broker* sea el único componente obligatorio. Es por eso que los componentes anteriormente mencionados que se construyen alrededor de ese *Context Broker* no tienen por qué ser necesariamente de FIWARE, sino que pueden pertenecer a terceros, según las necesidades de los desarrolladores.

El enfoque de FIWARE hacia la gestión de la información de contexto se justifica debido a que esta es la base de la nueva economía de datos y de la creación de nuevos e innovadores modelos de negocio en mercados formados por diferentes sectores que buscan una transferencia de datos fluida entre ellos. FIWARE permite que organizaciones en distintos dominios puedan intercambiar datos en una capa común de gestión de información de contexto, lo cual se hace especialmente útil en el caso de las ciudades inteligentes, dónde es probable que surjan nuevas iniciativas de negocio basadas en Smart Solutions en las que la fluidez de la transferencia de los datos puede ser un factor determinante para su éxito.

La importancia de la estandarización para el acceso a la información de contexto ha hecho que la labor de FIWARE haya sido reconocida por múltiples organizaciones estandarizadoras a nivel internacional como GSMA, CEF, TM Forum y ETSI entre otras.

Una vez introducido el marco tecnológico se da paso al siguiente capítulo, en el cual se explicarán las herramientas y recursos utilizados para el desarrollo del proyecto descrito en este documento. En la parte de desarrollo se hace uso del *Context Broker* de FIWARE anteriormente explicado y además se complementa su funcionalidad con herramientas de terceros ampliamente utilizadas en varios entornos que permiten procesar y visualizar los datos de contexto.

Como se ha comentado anteriormente, la solución desarrollada comparte con las herramientas de FIWARE (p.e. Wirecloud) el uso del estándar NGSI para la obtención de datos. Sin embargo, a diferencia de las soluciones existentes también se hace uso de modelos estándares de datos definidos que permiten uniformizar el tratamiento y representación de los datos.

Capítulo 3

Herramientas y recursos utilizados

En este capítulo se van a introducir las herramientas más importantes que se han utilizado para el desarrollo del presente proyecto. En capítulos anteriores se ha enmarcado este trabajo en un contexto tecnológico, el cual engloba las ideas más generales sobre las que se van a sustentar las herramientas y recursos que van a ser explicadas en este capítulo.

3.1. Interfaz NGSI

La estandarización llevada a cabo por FIWARE se materializa en el desarrollo de un API NGSI (Next Generation Service Interface) que está diseñado para gestionar el ciclo de vida completo de la información de contexto, incluidas actualizaciones, consultas, registros y suscripciones. FIWARE NGSI es la API expuesta por un FIWARE Context Broker, utilizada para la integración de componentes dentro de una plataforma y por aplicaciones para actualizar o consumir información de contexto.

SmartSantander posee una gran variedad de APIs, tanto propietarios como de terceros, el API NGSI de FIWARE es uno de ellos. Dicho API define principalmente los siguientes elementos:

- Un modelo de datos para la información de contexto, basado en un modelo simple de información que se basa en las entidades de contexto.

- Una interfaz de datos de contexto para el intercambio de información por medio de operaciones de consulta, suscripción y actualización.

- Una interfaz de disponibilidad de contexto para intercambiar información sobre cómo obtener información de contexto (actualmente se está discutiendo si separar ambas interfaces).

Como se ha comentado en el capítulo anterior, la definición del estándar NGSI ha variado a lo largo del tiempo. Actualmente FIWARE reconoce como estable la versión 2, referenciada como NGSI v2. A su vez se trabaja en la implementación del estándar NGSI-LD definido por la ETSI. A continuación se realiza una revisión de la principales características de la interfaz NGSI [7].

La interfaz NGSI permite el intercambio de información siguiendo un modelo de datos definido, el cual se compone de tres elementos principales: entidades, atributos y metadatos, tal como se muestra en la siguiente imagen.

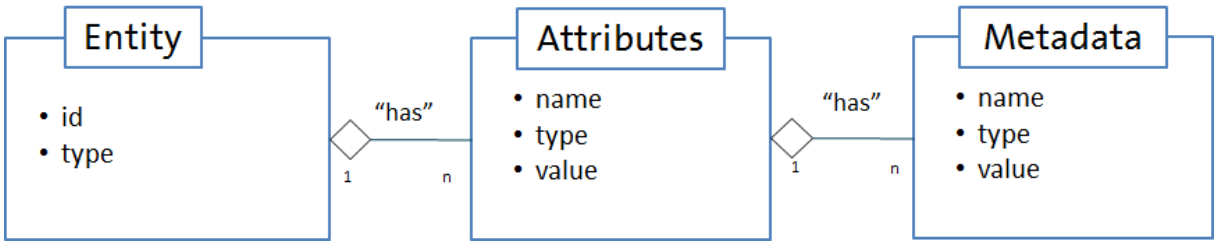

Figura 3.1: Modelo de datos genérico

Las entidades, también conocidas como entidades de contexto, se definen como objetos JSON y son la parte central en el modelo de información NGSI de FIWARE. Una entidad representa cualquier objeto físico o lógico, como por ejemplo sensores, personas, estancias, etc.

Al tener que realizarse un intercambio de información de contexto, se ha de poder asociar dicha información a la entidad que la ha obtenido, por tanto todas las entidades llevarán asociadas un identificador para este propósito. Además, el estándar NGSI define que cada entidad tendrá un tipo, el cual tiene como objetivo describir la naturaleza de lo que está representando la entidad. Por ejemplo, podría existir una entidad de contexto cuyo identificador sea *sensor-23* y que sea

del tipo *temperatureSensor*. Esto logra que cada entidad quede identificada de manera única por la combinación de su identificador y su tipo.

Cada entidad, aparte de ser identificable, posee ciertas propiedades llamadas atributos de contexto. En el modelo de información NGSI, estos atributos poseen un nombre, un tipo, un valor y unos metadatos:

- **Nombre del atributo**: describe que tipo de propiedad representa el valor del atributo de la entidad.
- **Tipo de atributo**: representa el tipo del valor del atributo de la entidad. NGSI tiene su propio sistema de tipos para valores de atributo, por lo que los tipos de valores NGSI no son los mismos que los tipos JSON.
- **Valor del atributo**: contiene los datos propiamente dichos y, de manera opcional, metadatos que describen las propiedades del valor del atributo como por ejemplo precisión, proveedor o marca de tiempo.

El último de los tres principales elementos de NGSI son los metadatos, los cuales pueden ser utilizados en distintos lugares. Uno de ellos es el que se ha visto anteriormente en el que se colocan de manera opcional como parte del valor de los atributos. De manera similar a lo que sucedía con los atributos, cada elemento de metadatos posee un nombre, un tipo y un valor:

- **Nombre de los metadatos**: describen la función de esos metadatos en el lugar donde están posicionados.
- **Tipo de los metadatos**: describe el tipo de valor NGSI del valor de metadatos.
- **Valor de los metadatos**: contiene los metadatos reales.

Se ha de tener en cuenta que NGSI no prevé que ciertos metadatos contengan en su interior más metadatos anidados.

Una vez que se ha explicado la terminología básica sobre el modelo de datos con el que trabajará NGSI, se procede a documentar los datos técnicos del API, los cuales serán necesarios para saber cómo acceder a él y qué se espera obtener al realizar una petición.

Para saber cómo se ha de procesar un documento es necesario conocer cual es su formato. Los tipos MIME (*Multipurpose Internet Mail Extensions*) son una forma estandarizada de indicar la naturaleza y el formato de un documento, archivo o conjunto de datos. El *payload* de una respuesta del API en la especificación NGSI v2 se basa en estos tipos MIME, concretamente en *application/json* y *text/plain*, este último para operaciones del tipo valor de atributo. Los clientes que emitan solicitudes HTTP con tipos de aceptación diferentes a estos, obtendrán un error `406 Not Aceptable`.

En la especificación también se define el modo de representación de las entidades y sus respectivos atributos, el cual se basa en un objeto JSON que sigue la estructura del modelo de datos genérico previamente definido. En las Figuras 3.2 y 3.3 se muestra un ejemplo visual de dicha representación.

```
{
  "id": "entityID",
  "type": "entityType",
  "attr_1": <val_1>,
  "attr_2": <val_2>,
  ...
  "attr_N": <val_N>
}
```

Figura 3.2: Representación entidad

```
{
  "value": <...>,
  "type": <...>,
  "metadata": <...>
}
```

Figura 3.3: Representación atributo

La gran variedad de parámetros que es capaz de contener una entidad hace que en ocasiones existan múltiples atributos, los cuales pueden tener distintos tipos. El tipo de dato que representa un atributo, o a unos metadatos, puede omitirse en las solicitudes al API. Cuando se omite en la creación de atributos/metadatos o en las operaciones de actualización, se usa un valor predeterminado para el tipo de dato según el valor:

- Si el valor es un *string*, entonces el tipo es `Text`
- Si el valor es un número, entonces el tipo es `Number`
- Si el valor es un booleano, entonces el tipo es `Boolean`

- Si el valor es un objeto o un *array*, entonces el tipo es `StructuredValue`
- Si el valor es *null*, entonces el tipo es `None`

Los metadatos de atributos pueden omitirse en las solicitudes, lo que significaría que no habría elementos de metadatos asociados al atributo. En este caso, en las respuestas, esta propiedad se establece como .

Además de los tipos de datos anteriormente comentados, existen otros tipos de atributos especiales, los más importantes son los que representan la fecha (`DateTime`) y los que se relacionan con la localización (`geo`).

Hay propiedades de las entidades que los clientes NGSI no pueden modificar directamente, pero que los servidores pueden representar para proporcionar información adicional. Desde el punto de vista de la representación, son como atributos regulares, con nombre, valor y tipo. Un ejemplo de este tipo de atributos son los relacionados con el tiempo (tipo `DateTime`). Los más importantes son los siguientes:

- **`dateCreated`**: Representa la fecha de creación en formato ISO 8601.
- **`dateModified`**: Representa la fecha de modificación en formato ISO 8601.
- **`dateExpires`**: Representa la fecha de expiración en formato ISO 8601. Es necesario precisar que la forma en que el servidor controla la caducidad es un aspecto específico de la implementación.

Por otro lado, en relación a los tipos de dato de localización, se define que las propiedades geoespaciales de una entidad también se pueden representar mediante atributos. Las implementaciones deben soportar dos tipos de sintaxis:

- **Simple Location Format**

  El formato de ubicación simple fue diseñado como un formato muy ligero para permitir a los desarrolladores y usuarios agregarlo de manera rápida y fácil a sus entidades existentes. Admite geometrías básicas (punto, línea, cuadro, polígono) y cubre los casos de uso típicos al codificar ubicaciones geográficas.

Un atributo de contexto que representa una ubicación codificada con el formato de ubicación simple debe tener un valor que sea una lista de coordenadas. De forma predeterminada, las coordenadas se definen utilizando el sistema de referencia de coordenadas `WGS 84`, con unidades de latitud y longitud en grados decimales. Dicha lista de coordenadas permite codificar la geometría especificada por el tipo de atributo y se codifican de acuerdo con las reglas específicas para cada tipo, las cuales se definen a continuación:

- **`geo:point`**: Representa un punto. El valor del atributo debe contener un *string* que contenga un par de latitud-longitud válido, separados por comas.
- **`geo:line`**: Representa una línea. El valor del atributo debe contener una lista de *string* de pares válidos de latitud y longitud. Debe haber al menos dos pares.
- **`geo:polygon`**: Representa una polígono. El valor del atributo debe contener una lista de *string* de pares válidos de latitud y longitud. Debe haber al menos cuatro pares, siendo el último idéntico al primero (por lo que un polígono tiene un mínimo de tres puntos reales).
- **`geo:box`**: Representa una región rectangular, que a menudo se utiliza para definir la extensión de un mapa o un área aproximada de interés.

Las geometrías circulares no están soportadas, ya que la literatura describe diferentes deficiencias para las implementaciones.

La Figura 3.4 muestra un ejemplo del tipo de geolocalización más utilizado (`geo:point`):

```
{
  "location": {
    "value": "41.3763726, 2.186447514",
    "type": "geo:point"
  }
}
```

Figura 3.4: Representación de una localización de tipo `geo:point`

Cabe señalar que el formato de ubicación simple no pretende representar posiciones complejas en la superficie de la Tierra, las cuales requieren de la altitud de un punto por ejemplo.

- **GeoJSON**

  Es un formato de intercambio de datos geoespaciales basado en JavaScript Object Notation (JSON). GeoJSON proporciona una mayor flexibilidad que permite la representación de altitudes de puntos o incluso formas geoespaciales más complejas, por ejemplo, múltiples geometrías.

  Un atributo de contexto que representa una ubicación codificada con GeoJSON debe ser del tipo `geo:json` y el valor del atributo debe ser un objeto GeoJSON válido (la longitud se antepone a la latitud en las coordenadas GeoJSON).

  La Figura 3.5 mostrada a continuación ilustra el uso de GeoJSON:

```
{
  "location": {
    "value": {
      "type": "Point",
      "coordinates": [2.186447514, 41.3763726]
    },
    "type": "geo:json"
  }
}
```

Figura 3.5: Representación de una localización de tipo GeoJSON

En base a la estructura de los datos y a los valores que pueden tomar ciertos campos, se ha de prever situaciones en las que haya errores. Para casos en los que algo haya ido mal, NGSI provee un objecto JSON el cual tiene un campo obligatorio (`error`) que contiene una descripción textual del error, y un campo opcional (`description`) que aportaría información adicional acerca de ese error.

Además, todas las implementaciones del servidor NGSI deben utilizar los siguientes códigos de estado HTTP entre otros:

- **`ParseError (400)`**: Si el JSON del *payload* no se puede parsear.

- **`BadRequest (400)`**: Para errores causados solo por la petición en sí, ya sea en los parámetros de URL o en el *payload*.
- **`NotFound (404)`**: Si no se encuentra el recurso solicitado en la petición.
- **`MethodNotAlowed (405)`**: Si el método de la petición no es permitido.
- **`TooManyResults (409)`**: Si se produce ambigüedad debida a que la petición puede referirse a varios recursos.
- **`ContentLengthRequired (411)`**: Si necesita que haya cierta longitud del contenido y no se provee.
- **`NoResourceAvailable (413)`**: Intento de exceder el límite del índice espacial en un campo de tipo localización (`geo`).
- **`Unprocessable (422)`**: Errores debidos a la combinación de la petición más el estado, pero no exclusivamente de la petición.

Otra de las funcionalidades que se recoge en la especificación del API NGSI es la llamada *Simple Query Language*, el cual es un lenguaje de consulta sencillo que proporciona una sintaxis simplificada para obtener datos de entidades que cumplen un conjunto de condiciones. Una consulta (*query*) está compuesta por una serie de instrucciones cada una de las cuales lleva asociada una condición. La consulta devuelve todas las entidades que cumplen esas condiciones.

Las instrucciones están compuestas por una ruta al atributo que se desea obtener información, un operador y un valor. Los operadores disponibles son: igual a (==), distinto a (!=), mayor que (>), menor que (<), mayor o igual que (>=), menor o igual que (<=) y la coincidencia de un patrón concreto (=). De este modo se podrían obtener ciertos datos, como por ejemplo los de los elementos que tengan un cierto valor de temperatura de un modo sencillo, `temperature==20`.

Otra de las más interesantes funcionalidades que provee NGSI es la posibilidad de especificar atributos en la URL de la petición, lo que permitirá definir operaciones que especifiquen una lista de ciertos atributos que se quieren incluir en la respuesta. Esta opción también está disponible para actuar sobre los metadatos. Por defecto si los atributos o los metadatos son omitidos, todos sus valores serán incluidos.

La información recogida en la documentación del API NGSI sirve para saber de qué modo se puede interactuar con ella. Una de las interacciones más importantes consiste en poder obtener los datos de contexto que han sido enviados a esa API. A pesar de que todos los datos lleguen en formato NGSI (modelo de información), pueden seguir diferentes modelos de datos, lo que sería un obstáculo para replicar una solución inteligente en diferentes ciudades o en diferentes servicios dentro de una ciudad.. En el siguiente capítulo se explicarán los modelos de datos que proporciona FIWARE para lograr una mayor homogeneidad en los datos.

3.2. Smart Data Models

Los *Smart Data Models* es una iniciativa colaborativa impulsada por FIWARE, TM Forum, UIDX y otras organizaciones y personas, que tiene como objetivo la adopción de una arquitectura de referencia y modelos de datos consistentes para el desarrollo de soluciones inteligentes [8].

Los *Smart Data Models* tienen un papel crucial porque definen los formatos de representación armonizados y la semántica que utilizarán las aplicaciones tanto para consumir como para publicar datos. La disponibilidad de modelos de datos ampliamente adoptados (estándar de facto) es clave para crear un mercado digital único y global de *smart solutions* interoperables y replicables en múltiples dominios. La portabilidad de los datos para que sean utilizados en diferentes aplicaciones es uno de los aspectos más atractivos de los *Smart Data Models*, los cuales están pensados para usarse donde se desee, pero con conformidad con FIWARE NGSI v2 y NGSI-LD.

Existen diversos sectores que pueden hacer uso de estos *Smart Data Models*, es por eso que en los repositorios donde se almacenan están organizados en dominios. Los cuales a su vez pueden contener subdominios, y éstos, contendrían los diferentes modelos de datos asociados. Los dominios a los que pueden pertenecer los *Smart Data Models* son los siguientes:

- **Smart Cities**: Engloba modelos de datos útiles en ciudades inteligentes.
- **Smart Environment**: Relacionado con el medio ambiente.
- **Smart Aeronautics**: Relacionado con la aeronáutica.
- **Smart Agrifood**: Relacionado sector agroalimentario.

- **Smart Robotics**: Engloba la robótica y otros sectores relacionados con ella.
- **Smart Destination**: Relacionado con los destinos turísticos.
- **Smart Manufacturing**: Relacionado con la fabricación. Este dominio se encuentra todavía en fases de desarrollo.
- **Smart Water**: Relacionado con la gestión del agua potable.
- **Smart Sensoring**: Relativo a los sensores.
- **Smart Energy**: Relacionado con la energía, tanto renovable como no renovable.
- **Cross sector**: Este dominio sirve para agrupar modelos de datos relacionados con otros dominios.

Uno de los dominios de mayor relevancia es el de *Smart Cities* debido a que existen numerosas fuentes de información de donde se puede extraer datos de contexto en las ciudades. Una de las principales barreras para conseguir una mejor explotación de estos datos es la inconsistencia en los modelos de datos, ya que esto bloquea la facilidad de integración de los mismos. El haber acordado entre diferentes comunidades, la definición común de modelos de datos de ciudades inteligentes, ha permitido a las empresas desarrollar soluciones que se adhieran a esta definición común y, por lo tanto, se fomente la interoperabilidad de los servicios.

Para lograr aprovechar y administrar de manera más eficiente esta información en las ciudades, se han desglosado los silos de datos para poder aplicar diferentes técnicas de tratamiento y análisis en conjuntos de datos agregados y garantizar así que la experiencia de los ciudadanos pueda ser optimizada en los diferentes servicios de la ciudad. Este desglosamiento de los silos de datos del dominio de *Smart Cities* se hace perceptible en los subdominios asociados, los cuales se enumeran a continuación:

  - **Building**: Para modelar edificios de la ciudad y su gestión.
  - **Parking**: Para modelar los estacionamientos disponibles de la ciudad.
  - **ParksAndGardens**: Para modelar parques, jardines y espacios verdes de la ciudad.
  - **PointOfInterest**: Para modelar los puntos de interés de la ciudad.

- **Ports**: Para modelar datos relativos a los puertos y barcos de la ciudad.
- **Streetlighting**: Para modelar el alumbrado público y todo su equipo de control hacia una iluminación urbana eficaz y energéticamente eficiente.
- **Transportation**: Para modelar datos relacionados con los diferentes medios de transporte disponibles en la ciudad.
- **UrbanMobility**: Para modelar la movilidad urbana.
- **WasteManagement**: Para modelar las principales entidades involucradas en la gestión de residuos.
- **Weather**: Para modelar datos meteorológicos.

Estos subdominios anteriormente comentados engloba varios *Smart Data Models*, los cuales modelan datos específicos dentro de cada uno de ellos. Debido a que los contextos que pueden modelarse cada vez son mayores, FIWARE permite la creación por parte de los usuarios de nuevos modelos de datos, los cuales serán revisados por esta organización y determinarán si es válida su puesta en funcionamiento.

3.2.1. Definición de modelos de datos

Todos los *Smart Data Models*, aunque modelen diferentes datos, han de seguir una serie de pautas generales para que puedan ser utilizados de la misma manera por todos los servicios que quieran hacer uso de ese modelado de información. A continuación, se listan un conjunto de directrices para definir nuevos modelos de datos.

- **Sintaxis**:
  - Utilizar términos en inglés, preferiblemente inglés americano.
  - Para los nombres de atributos utilizar el estilo *camel case* (camelCase).
  - El nombre del tipo de las entidades debe empezar con mayúscula.
  - Utilizar nombres y no verbos para los atributos de tipo Propiedad.

  - Evitar los plurales en los nombres de los atributos, pero indicar claramente cuándo se esté haciendo referencia a una lista de elementos (Ejemplo: *category*)
  - Todas las propiedades de primer nivel definidas en el esquema JSON deben tener un atributo de descripción.

- **Reutilización**: Es recomendable verificar si alguno de los atributos que se van a añadir existe en otro modelo de datos ya definido, para así reutilizarlos. Visitar la página *schema.org* para intentar encontrar un término similar con la misma semántica, y cuando sea posible reutilizar los tipos de datos definidos ahí, como por ejemplo *Text*, *Number*, *DateTime*, *StructuredValue*, etc.

- **Definición de atributos**:

  - Enumerar los valores permitidos para cada atributo. En términos generales, es una buena idea dejarlo abierto para que las aplicaciones amplíen la lista, siempre que el nuevo valor no esté semánticamente cubierto por ninguno de los existentes. El valor *null* debe evitarse, ya que está prohibido en NGSI-LD.
  - Indicar claramente qué atributos son obligatorios y cuáles son opcionales. Los atributos mínimos requeridos harán que los modelos de datos sean más flexibles para que otros los usen.
  - Utilizar el prefijo *date* para nombrar atributos que representen fechas.
  - NGSI v2 tiene atributos internos creados por el sistema, los cuales son *dateCreated* y *dateModified*. De un modo análogo NGSI-LD tiene *createdAt* y *modifiedAt*. Estos atributos no deben incluirse en la definición del modelo de datos (spec.md y *schema.*) pero pueden aparecer en las cargas útiles de los ejemplos incluidos.
  - Cuando sea necesario, definir atributos adicionales para capturar con precisión todos los detalles sobre las fechas. Por ejemplo, para indicar la fecha en la que se entregó un pronóstico del tiempo, un atributo llamado *dateIssued* puede ser utilizado.
  - Cuando un valor sea un número, si es relevante es recomendable incluir límites de rango si existiesen.
  - Utilizar valores entre 0 y 1 para cantidades relativas, las cuales representen valores de atributos como por ejemplo *relativeHumidity* o *precipitationProbability*.

- **Atributos dinámicos**: En NGSI v2 utilizar un atributo de metadatos llamado *timestamp* para capturar el instante de actualización de un atributo dinámico. Tener en cuenta que

ésta es la fecha real en la que se obtuvo el valor medido, y esa fecha puede ser diferente a la fecha en el que se actualizó el atributo de la entidad digital (atributo de metadatos denominado *dateModified* según NGSI v2). Esto se debe a que normalmente puede haber retrasos, especialmente en las redes de IoT que entregan datos solo en intervalos de tiempo específicos.
En NGSI-LD utilizar la propiedad *observedAt* para transmitir marcas de tiempo.

- **Modelado de localización**: Utilizar los atributos *address* y *location* para definir la dirección y geolocalización respectivamente.
- **Control de versiones**: El proyecto de modelos de datos de FIWARE y TM Forum tiene como objetivo mantener la compatibilidad con versiones anteriores, sin embargo, inevitablemente se producirán algunas incompatibilidades con el tiempo. Los proveedores de datos pueden optar por etiquetar las Entidades con un atributo *schemaVersion* para que los consumidores de datos puedan comportarse en consecuencia.

3.3. Elastic stack

En capítulos anteriores ya se ha evidenciado la importancia de los datos para aportar un mayor valor a los potenciales servicios que puedan ofrecerse. Es por esto que a medida que han avanzado las técnicas de obtención de estos datos, también lo han hecho las herramientas para permitir un tratamiento de los mismos para así poder extraer su máximo potencial.

ELK Stack, era el nombre con el que se conocía a la pila que se compone de proyectos de código abierto que toman datos de cualquier fuente y formato, permitiendo buscar, analizar y visualizar esos datos en tiempo real. El acrónimo ELK hacía referencia a cada uno de los principales componentes de la herramienta: Elasticsearch, Logstash y Kibana. En el año 2015 se introdujo una familia de agentes de datos de propósito único y livianos, los cuales se llamaron Beats. Al conjunto de estos cuatro componentes se le dio el nombre de Elastic Stack [9].

La pila de Elastic en sus inicios se ofrecía como una solución a la gestión y tratamiento de los *logs*, los cuales son registros de los eventos que afectan a ciertas máquinas. La grabación de estos eventos en archivos o en bases de datos proporciona una visión acerca del comportamiento de sistemas y programas que se tienen desplegados. Elastic Stack comenzó aportando un análisis más profundo de estos archivos, así como una visualización de ellos, entre otras funcionalidades.

El gran potencial de análisis, procesamiento y visualización de los datos de Elastic Stack permitió extender su uso más allá de los *logs*, logrando así abarcar el tratamiento de datos de cualquier fuente y formato. Actualmente es una de las soluciones más ampliamente utilizadas para la representación de datos.

En la siguientes secciones se realizará una revisión de los aspectos más importantes de los principales componentes que componen el Elastic Stack.

3.3.1. Elasticsearch

En el panorama tecnológico actual, la capacidad de búsqueda es una de las principales funcionalidades necesarias en todas las aplicaciones. Esta necesidad puede ser satisfecha mediante Elasticsearch, el cual posee además otras características adicionales que se complementan con esta.

Elasticsearch es un motor de búsqueda de código abierto construido sobre Apache Lucene. Proporciona búsquedas y análisis casi en tiempo real para todo tipo de datos, tanto si se posee texto estructurado o no estructurado, datos numéricos o geoespaciales. Esto es debido a que Elasticsearch puede almacenar estos datos e indexarlos de manera eficiente para así lograr realizar búsquedas en poco tiempo.

Lucene es una librería de búsqueda rápida. Para aprovechar todo su potencial se debe de trabajar en Java e integrar Lucene directamente en la aplicación que se vaya a desarrollar. Además, su alta complejidad hace que los desarrolladores deban tener un alto grado de conocimiento sobre esa librería y sobre cómo funciona.

Elasticsearch también está escrito en Java y usa Lucene internamente para toda su indexación y búsqueda. Su objetivo principal es facilitar la búsqueda de texto completo. Esto lo consigue ocultando las complejidades de Lucene detrás de una API REST simple y coherente.

Antes de entrar en detalle acerca de las funcionalidades que provee Elasticsearch se hace una revisión de los términos generales más comúnmente utilizados en este entorno:

- **Nodo**: Es una sola instancia de Elasticsearch ejecutándose en una máquina.

- **Clúster**: Es el nombre único bajo el cual uno o más nodos están conectados entre sí.
- **Documento**: Elasticsearch almacena datos como documentos, los cuales son objetos JSON que contiene los datos reales en pares clave-valor (*key-value*). Cada documento correlaciona un conjunto de claves (nombres de campos o propiedades) con sus valores correspondientes (textos, números, booleanos, fechas, geolocalizaciones u otros tipos de datos).
- **Índice**: Un índice en Elasticsearch es una colección de documentos relacionados entre sí.

Elasticsearch no solo facilita la búsqueda con respecto a Lucene, sino que posee además otras muchas características que justifican su utilización. A continuación se explican algunas de las más importantes:

- **Distribuido y altamente escalable**: Desde su lanzamiento, Elasticsearch siempre ha tenido una naturaleza distribuida. Fue diseñado para ser escalado horizontalmente y no verticalmente. Este tipo de escalabilidad permite que se comience a desarrollar una aplicación con un clúster de Elasticsearch de un solo nodo y, a medida que el número de usuarios aumente, ese clúster podrá escalarse a cientos o miles de nodos sin tener que preocuparse por las complejidades internas que conlleva la computación distribuida, el almacenamiento distribuido y las búsquedas.
- **Alta disponibilidad**: La replicación de datos significa tener múltiples copias de datos en un clúster. Esta función permite a los usuarios crear clústeres de alta disponibilidad manteniendo más de una copia de los datos. Elasticsearch permite la emisión de un simple comando para que automáticamente se creen copias redundantes de los datos, obteniendo así una mayor disponibilidad y evitando su pérdida en caso de que se produzca algún fallo en la máquina que los almacena.
- **Orientado a documentos**: El almacenamiento de objetos con estructuras de datos complejas (como fechas, geolocalizaciones y otros objetos) es poco útil en bases de datos relacionales, ya que se tendría que conseguir que esos objetos complejos se adapten al esquema de la tabla en la que se están almacenando. Elasticsearch es orientado a documentos, es decir, que almacena directamente objetos (documentos). Además, también indexa el contenido de cada uno de ellos para hacer que se puedan buscar más

fácilmente, lo que se traduce en velocidad de búsqueda. Elasticsearch utiliza JSON como formato de serialización de documentos.

- **Query DSL**: Elasticsearch proporciona Query DSL (Domain Specific Language) basada un JSON para escribir y leer *queries* de una manera sencilla. Esto permite a los desarrolladores que no conocen la compleja sintaxis de consulta de Lucene escribir *queries* complejas en Elasticsearch.

- **Basado en REST**: Elasticsearch se basa en la arquitectura REST y proporciona *endpoint* no solo para realizar operaciones CRUD a través de llamadas de API HTTP, sino también para permitir a los usuarios realizar tareas de monitorización de clústeres utilizando APIs REST. Los *endpoints* de REST también permiten a los usuarios realizar cambios en la configuración de índices y clústeres de forma dinámica, en lugar de enviar manualmente actualizaciones de configuración a todos los nodos de un clúster editando el archivo `elasticsearch.yml` y reiniciando el nodo. Esto es posible porque cada recurso (índice, documento, nodo, etc.) en Elasticsearch es accesible a través de un URI simple.

Además de todas estas funcionalidades, Elasticsearch permite también la definición de plantillas de índices, las cuales son formas de decirle cómo configurar un índice cuando se crea. Las plantillas se configuran antes de que se cree un índice y después de su creación (ya sea manualmente o mediante la indexación de un documento), se puede configurar esa plantilla para definir las propiedades de ese índice. De todas las posibles configuraciones de plantillas, una de las más comunes es la definición del tipo de dato que se va a almacenar dentro de un campo de Elasticsearch, lo cual es especialmente útil para campos que contengan geolocalizaciones.

Otro de los aspectos más importantes a tener en cuenta es la ejecución de Elasticsearch, el cual se iniciará en dos puertos: 9200 y 9300. El primero es utilizado para crear conexiones HTTP, mientras que el segundo es utilizado para crear conexiones TCP a través de un cliente JAVA y para la interconexión del nodo dentro de un clúster. Debido a esto se aprecia que la comunicación con Elasticsearch es diferente dependiendo del lenguaje con el que se trate de hacer.

Debido a que Elasticsearch está desarrollado en Java, viene con dos clientes, *Node client* y *Transport client*, los cuales se comunican con el clúster de Elasticsearch sobre el puerto 9300 utilizando el protocolo de transporte nativo de Elasticsearch. Para que esta comunicación

funcione correctamente el cliente Java debe ser de la misma versión de Elasticsearch que los nodos, de lo contrario, es posible que no puedan entenderse.

Por otro lado, Elasticsearch proporcionar un API REST con JSON sobre HTTP que permite la interoperabilidad de lenguajes en comparación con la API básica implementada en Java de Lucene [10]. De este modo, el resto de lenguajes de programación pueden comunicarse con Elasticsearch utilizando el puerto 9200 de ese API REST. Para permitir dicha comunicación, Elasticsearch proporciona clientes oficiales para varios lenguajes: Groovy, JavaScript, .NET, PHP, Perl, Python y Ruby. Además existen numerosos clientes e integraciones proporcionados por la comunidad.

Elasticsearch es la pieza fundamental del Elastic Stack, debido a que es el encargado de almacenar los datos y realizar búsquedas rápidas sobre ellos en el caso de que sean solicitados. Sus funcionalidades se limitan principalmente a las comentadas, por lo que es muy común la utilización de Elasticsearch junto con el resto de módulos que forman el Elastic Stack. En las siguientes secciones se hará una revisión de estos módulos explicando sus funcionalidades principales y su interacción con Elasticsearch.

3.3.2. Kibana

El principal objetivo del almacenamiento de los datos en Elasticsearch, o en cualquier base de datos en general, es la de disponer de ellos en el futuro para hacer el uso que se desee de los mismos. Las ingentes cantidades de datos que pueden almacenarse imposibilitan a los usuarios obtener información de ellos con cierta facilidad. Es por eso que se han desarrollado herramientas, como Kibana, para permitir mostrar datos de manera visual, de modo que un usuario pueda hacerse una idea de lo que representan con tan solo mirar una gráfica.

Kibana es una aplicación de *frontend* de uso libre que se encuentra sobre el Elastic Stack y proporciona capacidades de visualización de datos y de búsqueda para los datos indexados en Elasticsearch. También actúa como interfaz de usuario para monitorizar, gestionar y asegurar un clúster del Elastic Stack. Kibana ha llegado a ser la ventana al propio Elastic Stack debido a que ofrece un portal para que los usuarios y las empresas puedan acceder a representaciones realmente informativas de sus datos.

La búsqueda y visualización de datos en Kibana se basa en actuar sobre uno o varios índices de Elasticsearch. Dichos índices agrupan una colección de documentos relacionados entre sí, y son creados en el momento de la inyección de datos (en formato de documentos) en Elasticsearch. La interfaz de Kibana permite a los usuarios buscar datos en índices de Elasticsearch y a continuación visualizar los resultados a través de gráficos estándar o aplicaciones integradas. Los usuarios pueden elegir entre diferentes tipos de gráficos, cambiar las agregaciones de números y filtrar segmentos específicos de datos para conseguir una visualización personalizada de los datos.

Kibana requiere de un patrón de índice, de aquí en adelante referido como *index pattern*, para acceder a los datos de Elasticsearch. Un *index pattern* identifica uno o más índices de Elasticsearch permitiendo así la exploración de los datos contenidos en esos índices. Kibana busca nombres de índice que coincidan con el patrón especificado para así poder representarlos.

Las representaciones gráficas que permite Kibana vienen normalmente precedidas por una exploración de los datos para conocer cuantos datos hay, por qué campos están compuestos y de qué tipo son. Para explorar los datos Kibana cuanta con la herramienta Discover, la cual permite acceder a cada uno de los documentos de Elasticsearch contenidos en índices que coincidan con el *index pattern* seleccionado y obtener detalles a nivel de campo sobre ellos. Además permite buscar datos y filtrar resultados de búsqueda y ver los eventos que ocurrieron justo antes y después de un documento. Discover muestra de manera predeterminada los datos de los últimos 15 minutos, aunque permite el ajuste del rango de este filtro de tiempo para visualizar los datos en las fechas que se decida.

Kibana, además de descubrir los datos, permite generar diferentes visualizaciones y análisis para mostrarlos de una manera agradable. Los formatos de representación que permite Kibana son variados, a continuación se añade una lista con los más importantes:

- **Charts**: Los gráficos de línea, área y barras permiten representar los datos en los ejes X e Y. También está la opción disponible de mostrar gráficos circulares para representar porcentajes de la contribución de cada fuente a un total.

- **Metrics**: Esta visualización de métricas muestra un solo número para cada agregación. Permite mostrar métricas de recuento, promedio, suma, mínimo, máximo, desviación estándar, percentiles, etc.

- **Data tables**: La representación en tablas es una de las formas más comunes de expresar ciertos datos. Esta visualización de Kibana permite la configuración de las tablas de datos para capturar un momento en el tiempo, o su sincronización con los datos en tiempo real para una vista dinámica y actualizada de lo que está sucediendo.

- **Kibana Lens**: Es una interfaz de usuario intuitiva y fácil de usar que simplifica el proceso de creación de visualizaciones de datos. Su simplicidad reside en que para la creación de representaciones visuales, el usuario solo tiene que seleccionar los campos que quiere visualizar, arrastrarlos y soltarnos en un panel para que se genere automáticamente una visualización. Esta herramienta es verdaderamente útil para usuarios que están iniciándose en la generación de visualizaciones con Kibana.

- **Time Series Visual Builder (TSVB)**: Consta de diferentes opciones de visualización, aunque la más utilizada es la representación de series temporales. Para este caso de uso, TSVB requiere un campo de fecha para poder mostrar los valores en la gráfica respecto a un eje temporal. TSVB es compatible con la mayoría de las agregaciones de métricas de Elasticsearch, múltiples tipos de visualización, funciones personalizadas y algunas operaciones matemáticas.

- **Maps**: Permite representar en un mapa geolocalizaciones para obtener una representación visual del lugar de donde proviene la información. Los datos geográficos comprenden más que solo latitudes y longitudes, cada punto del mapa puede contener métricas, una marca de tiempo y metadatos adicionales. Además, no solo se permite la visualización de puntos en un mapa, sino que también posibilita la opción de unir puntos para la representación de líneas.

  Esta herramienta brinda la posibilidad de mostrar todos los datos en un solo mapa por medio de la creación de múltiples capas e índices, y dado que las capas están en el mismo mapa, se puede buscar y filtrar en todas ellas en tiempo real.

- **Canvas**: Permite extraer datos directamente de Elasticsearch para combinarlos con colores, imágenes, formas y texto, para que se puedan crear pantallas dinámicas de varias páginas agradables visualmente.

Debido a la gran variedad de visualizaciones disponibles es muy común que los usuarios quieran agrupar algunas de ellas para mostrarlas como parte de un conjunto. Para proveer esta funcionalidad Kibana provee los llamados *dashboards*, los cuales permiten recoger datos de uno o varios *index patterns* y agruparlos en una colección de visualizaciones para represetarlos en un solo panel. Esto aporta una mayor claridad de los datos dando la posibilidad de tener en una sola pantalla las diferentes visualizaciones referidas a un tipo de datos.

La posibilidad de organizar, cambiar el tamaño y editar el contenido permite la creación de *dashboards* adaptados a las preferencias del usuario. Normalmente, estas colecciones de visualizaciones se crean para ser compartidas y que varios usuarios puedan hacer uso de ellas. Es por esto que Kibana habilita la opción de compartir *dashboards* de diferentes maneras, las más comunes son embeberlos como un *iframe* en una página web y generar un enlace directo a ese *dashboard* en Kibana.

La posibilidad de interactuar con los *dashboards* hace que la experiencia de usuario aumente de manera significativa. Es por eso que Kibana proporciona herramientas de visualización como *Markdown*, la cual brinda la opción de crear mensajes de texto y colocarlos en el los *dashboards*. Dichos mensajes de texto pueden contener URLs que dirijan a imágenes alojadas en la web o incluso a otros *dashboards*.

Otra visualización de Kibana que permite la interacción con los usuarios es *Controls*, la cual permite agregar entradas interactivas a los *dasboards*. Con esta herramienta se pueden crear entradas de dos tipos: un menú desplegable que permite a los usuarios filtrar contenido seleccionando una o más opciones de una lista y unos controles de selección de rango que permiten a los usuarios filtrar contenido dentro de un rango de números.

Además de las opciones de exploración y visualización anteriormente comentadas, Kibana dispone de una serie de extensiones. Una de estas funcionalidades es la de alertas, mediante la cual permite detectar condiciones complejas dentro de diferentes aplicaciones de Kibana y activar acciones cuando se cumplen esas condiciones.

Otra de las funcionalidades que proporciona es la de *Machine Learning*, la cual cobra verdadera utilidad cuando los conjuntos de datos aumentan en tamaño y complejidad dificultando la detección de problemas o anomalías. A pesar de que Kibana es de código abierto, algunas funcionalidades extra como ésta de *Machine Learning* son de pago.

Kibana ofrece también interfaces de usuario para gestionar todo lo relacionado con el Elastic Stack como por ejemplo índices, clústeres, licencias, configuraciones de interfaces de usuario, *index patterns* y más. Además, Kibana posee una serie de herramientas de desarrollo (*Dev Tools*) que pueden utilizarse para interactuar con los datos. Dentro de ellas destaca la consola que permite comunicarse con el API REST de Elasticsearch, de modo que se pueden mandar peticiones y visualizar las respuestas, ver la documentación del API y obtener el historial de peticiones entre otras cosas.

3.3.3. Logstash

El almacenamiento de los datos y las capacidades de búsqueda de Elasticsearch junto con el potencial de representación de datos de Kibana forma un sistema muy completo en el que se puede extraer valor de los datos analizados. A pesar de que estos dos componentes permiten tener un entorno apropiado para tratar casi todo tipo de datos, el equipo de Elastic desarrolló otro componente para facilitar la recopilación de los datos y su inserción en Elasticsearch. Dicho componente es Logstash.

Los datos que pueden almacenarse en Elasticsearch son de diferente naturaleza, debido a que son múltiples las fuentes de las que pueden provenir dependiendo del escenario en el que se esté utilizando este software. El acceso a esas fuentes de información y su posterior inserción en Elasticsearch puede resultar un proceso tedioso y complicado en muchos casos. Herramientas como Logstash nacen para facilitar esta tarea.

Logstash es un motor de recopilación de datos de código abierto que permite la obtención y transformación de datos para una inserción eficiente en los destinos que se desee. Actualmente, Logstash es uno de los principales componentes del Elastic Stack, en la mayoría de los casos interactúa directamente con Elasticsearch, en donde inserta los datos recopilados y transformados.

Antes de explicar en profundidad las funcionalidades de Logstash, en la Figura 3.6 se ilustra la interacción de los componentes del Elastic Stack mencionados hasta el momento. El flujo de los datos comenzaría con la recopilación de la información por parte de Logstash, seguido por su inserción en Elasticsearch y acabando con su representación gráfica mediante Kibana [11].

Figura 3.6: Flujo de datos Logstash, Elasticsearch y Kibana

Logstash nació como una herramienta de código abierto desarrollada para manejar la transmisión de una gran cantidad de *logs* provenientes de múltiples fuentes. De este modo suplía la necesidad de recopilar y transformar los diferentes tipos de *logs* que estaban siendo generados en máquinas, y que aportaban información acerca de los eventos que estaban sucediendo en ellas, para su almacenamiento en los destinos deseados [12]. La evolución de Logstash, mediante el trabajo del equipo de Elastic, ha logrado extender esa naturaleza inicialmente orientada a *logs* para acabar proporcionando un *framework* para la recopilación, centralización, análisis y almacenamiento y de muchos otros tipos de datos. Después de incorporarse al Elastic Stack, Logstash se convirtió en un elemento fundamental de la pila.

El motor de procesamiento y recopilación de datos que implementa Logstash está basado en *plugins*, los cuales permiten configurarlo fácilmente para recopilar, procesar y reenviar datos en muchas arquitecturas diferentes. El procesamiento se organiza en uno o varios *pipelines*, es decir, en caminos que siguen los flujos de datos. La estructura de cada uno de estos *pipelines* está generalmente formada por tres componentes, los cuales se representan a continuación.

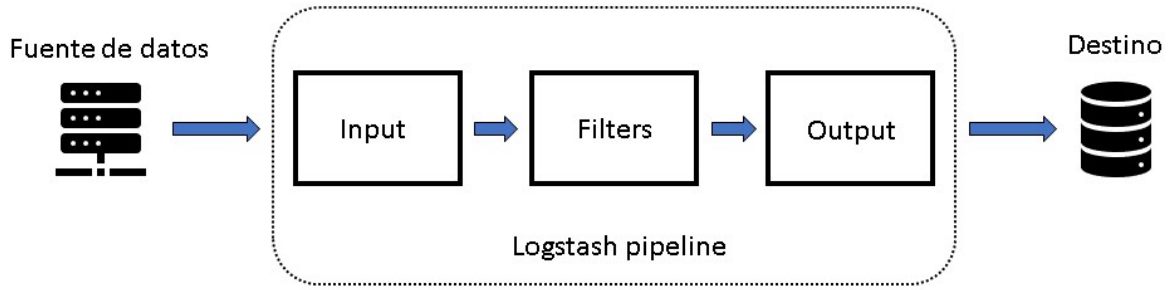

Figura 3.7: Componentes de Logstash

Tal como se aprecia en la Figura 3.7, los componentes que forman un *pipeline* de Logstash son las entradas (*input*), los filtros (*filters*) y las salidas (*output*). Cada uno de ellos tiene una función específica y puede hacer uso de diversos *plugins* disponibles. A continuación se hace una revisión de los aspectos más importantes de estos componentes.

- **Entradas**

  Los datos a menudo se encuentran en silos de información en multitud de sistemas y con un diferentes formatos. Logstash admite una gran variedad de entradas que tienen como principal objetivo la extracción de datos de una gran cantidad de fuentes comunes al mismo tiempo. Posteriormente esa información extraída se traduce en la generación de eventos, cada uno de los cuales es una sola unidad de información que contiene un *timestamp* y más datos adicionales. Estos eventos son la forma de comunicación de Logstash.

  Los lugares más comunes de los que Logstash recopila información son:

  - ***Logs* y métricas**: Esta fue la primera funcionalidad que aportó Logstash, permitiendo la captura de *logs* y de métricas generadas por diferentes aplicaciones.
  - **La web**: Logstash permite además la transformación de peticiones HTTP en eventos, de modo que se hace posible consumir datos de servicios web. Además, habilita la creación de eventos mediante el sondeo de *endpoints* bajo demanda.
  - **Almacenes de datos**: La posibilidad de extraer información de bases diferentes bases de datos supone una gran ventaja en el caso de que se requiera trabajar con datos ya almacenados sobre los cuales se podrán realizar consultas.
  - **Sensores e IoT**: Logstash también puede ser usado para explorar otro tipo de datos, como son los generados por dispositivos IoT interconectados.

  Para permitir la recopilación de datos de los tipos de fuentes anteriormente comentados, Logstash proporciona una gran variedad de *plugins*. A continuación se listan algunos de los más utilizados:

  - **File**: Para leer archivos y transmitir sus datos en eventos.
  - **Syslog**: Escucha en el puerto 514 para leer los mensajes de *syslog* como eventos.
  - **HTTP**: Esta entrada permite recibir eventos de una o varias líneas a través de HTTP o HTTPS. Las aplicaciones pueden enviar una solicitud HTTP al *endpoint* iniciado por esta entrada y Logstash la convertirá en un evento para su procesamiento posterior.

  - **HTPP Poller**: Permite realizar llamadas periódicas a una API HTTP y decodificar su respuesta. Hace uso del campo *schedule* en el que se puede definir cada cuanto se van a realizar esas llamadas.
  - **Elasticsearch**: Permite obtener resultados de una *query* realizada a un clúster de Elasticsearch.
  - **Beats**: Permite recibir eventos del componente Beats perteneciente al Elastic Stack. Esta funcionalidad se describirá en más detalle en la siguiente sección.

- **Filtros**

  Los filtros son componentes de procesamiento intermedios en un *pipeline* de Logstash. Se sitúan justo después del proceso de extracción de los datos, y antes de que esos datos lleguen al destino donde van a ser almacenados. Los filtros de Logstash parsean cada evento, identifican los campos con nombre para crear la estructura y los transforman para que converjan en un formato común y más sencillo.

  Logstash transforma y prepara de forma dinámica los datos recolectados independientemente de su formato o complejidad. Para ello proporciona una gran librería de *plugins*, algunos de los más comunes son listados a continuación:

  - **Mutate**: Permite realizar transformaciones generales en los campos de eventos. Las acciones más comunes son cambiar el nombre, eliminar, reemplazar y modificar, entre otras.
  - **Date**: Analiza las fechas de los campos para usarlas como *timestamp* (marca de tiempo) de Logstash para un evento. En ausencia de este filtro, Logstash elegirá un *timestamp* en función de la primera vez que vea el evento (en el momento de la entrada), en el caso de que el *timestamp* aún no esté configurado en el evento.
  - **Grok**: Permite parsear datos de eventos no estructurados en campos con cierta estructura. Esta herramienta resulta muy apropiada para tratar con *logs*.
  - **GeoIP**: Añade información sobre la ubicación geográfica de las direcciones IP.
  - **HTTP**: Proporciona integración con servicios web externos/API REST.

  Logstash, además de proporcionar diversos filtros, permite la posibilidad de combinarlos con instrucciones condicionales para realizar una acción en un evento si cumple con ciertos criterios.

- **Salidas**

  Las salidas es la última etapa del *pipeline* de Logstash. Una vez que se han obtenido y transformado los datos solo falta su almacenamiento para que puedan ser accesibles en otro momento. Logstash ofrece una gran variedad de salidas para enrutar los datos hacia donde se desee. Esto aporta una gran flexibilidad a la hora de hacer uso de ellos en el futuro.

  Los *plugins* de salida más importantes son los que se listan a continuación, pero, además de estos, Logstash ofrece una gama mucho más amplia para adaptarse a un gran número de posibilidades de almacenamiento.

  - **Elasticsearch**: Este *plugin* almacena los datos en Elasticsearch. Permite la definición de la dirección dónde se encuentra el servidor, así como el índice en el que se almacenarán esos datos.

    Esta es una de las opciones más populares debido a que se estaría constituyendo una comunicación únicamente entre componentes del Elastic Stack, los cuales están específicamente desarrollados para facilitar esa interacción.
  - **Influxdb**: Proporciona un almacenamiento de los datos en una base de datos orientada a series temporales como es InfluxDB.
  - **File**: Esta salida escribe eventos en archivos locales. De forma predeterminada, cada línea de escritura representa un evento descrito en formato JSON, aunque esto puede modificarse.
  - **Email**: Utilizado para habilitar el envío automático de un correo electrónico cuando se reciba una salida de datos.
  - **HTTP**:Este *plugin* permite enviar eventos a *endpoints* HTTP o HTTPS.

Además de lo comentado anteriormente, Logstash ofrece los llamados *codec plugins*, los cuales son básicamente filtros que pueden operar como parte de una entrada o una salida. Los más comunes son, por un lado el filtro *csv codec* el cual recoge datos CSV, los parsea y los transmite, y por otro lado el filtro *json codec* el cual puede actuar en una entrada para transformar mensajes JSON en eventos y en una salida para realizar el proceso contrario.

Logstash provee un marco de trabajo en el que están disponibles más de 200 *plugins*. Mezcla, combina y orquesta diferentes entradas, filtros y salidas para trabajar en armonía con el pipeline. Además de esto, permite la opción de crear *plugins* personalizados para aplicaciones específicas, lo que permite definir *pipelines* específicos para determinadas aplicaciones.

Logstash define dos tipos de archivos de configuración:

- **Archivos de configuración de *pipeline***

  En este tipo de archivos se definen los *plugins* que van a utilizarse para las entradas, filtros y salidas de modo que permitan la definición del *pipeline* específico que se desee crear. Estos archivos deben crearse con una extensión `.conf` y ser ejecutados con un comando de Logstash.

- **Archivos de configuración generales**

  Estos archivos ya están definidos en la instalación de Logstash y pueden ser modificados según las necesidades del usuario. Estos archivos son los siguientes:

  - **`logstash.yml`**: Contiene *flags* de configuración de Logstash, lo cual facilita la ejecución de comandos ya que no sería necesaria la definición de ciertos *flags* si ya se definieron en este archivo de configuración. Se ha de tener en cuenta que cualquier *flag* que se establezca en la línea de comandos anula la configuración correspondiente en este archivo `logstash.yml`.
  - **`pipelines.yml`**: Contiene el *framework* y las instrucciones para la ejecución de múltiples *pipelines* en una sola instancia de Logstash.
  - **`jvm.options`**: Contiene *flags* de configuración de JVM. Se utiliza este archivo para establecer valores máximos y mínimos del espacio total del *Heap*.
  - **`log4j2.properties`**: Contiene la configuración predeterminada para la librería `log4j 2`, la cual es la solución mayormente adoptada para la generación de *logs*.

En definitiva, Logstash es una potente herramienta para la recolección, transformación e inserción en otros lugares de los datos. Se definen *pipelines* que establecerán cómo se va a actuar sobre los datos, es decir, de dónde se van a extraer, cómo se van a transformar y dónde se van a almacenar. A pesar de la gran flexibilidad y potencia de Logstash, existen otras formas de obtención a inserción de datos, como es el caso del componente Beats, también perteneciente al Elastic Stack.

3.3.4. Beats

Logstash se suele utilizar para inyectar datos a gran escala en Elasticsearch, pero existen ciertas aplicaciones que no necesitan de todas las funcionalidades que provee Logstash y que se podrían ejecutar perfectamente con un software más liviano. Aquí es donde entra en juego Beats.

Beats es una plataforma gratuita y abierta para agentes de datos con un solo propósito. Dichos agentes tienen como objetivo la recopilación de datos. Se suelen instalar en los servidores o se despliegan como funciones, y después centralizan los datos en Elasticsearch. En la Figura 3.8 se muestra dónde se posicionaría Beats en el Elastic Stack junto con el resto de componentes.

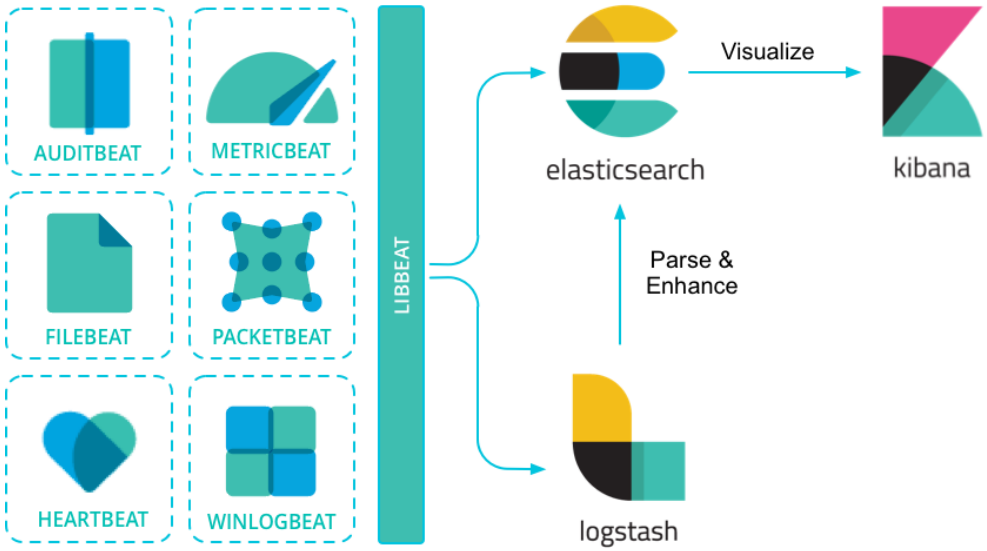

Figura 3.8: Flujo de los datos del Elastic Stack completo

Como se puede observar, Beats puede enviar datos directamente a Elasticsearch o a través de Logstash, donde puede procesar y mejorar aún más los datos, antes de visualizarlos en Kibana. En el caso de que Beats enviase datos a Logstash y éste a Elasticsearch, se haría uso del *plugin* de entrada de Logstash llamado *beats*, que, tal como se comentó en el anterior capítulo, es el encargado de recibir eventos emitidos por Beats.

Existen diferentes tipos de agentes Beats dependiendo del tipo de datos que se quiera capturar. Es por esto que cada vez se van generando más agentes para ajustarse a todos los tipos de datos posibles. Actualmente, algunos agentes Beats se encuentran en fases de desarrollo. Los más importantes y completamente testados son los que se muestran en Figura 3.8 y cuya funcionalidad se describe brevemente en la siguiente lista:

- **Filebeat**: Se instala como un agente en los servidores y es utilizado para obtener los *logs* generados.
- **Metricbeat**: Se encarga de recopilar periódicamente métricas del sistema operativo y de los servicios que se ejecutan en el servidor.
- **Packetbeat**: Es un analizador de paquetes de red en tiempo real que se usa para proporcionar un sistema de análisis de rendimiento y monitorización de aplicaciones.
- **Winlogbeat**: Se encarga de leer *logs* de eventos utilizando las APIs de Windows, luego los filtra según los criterios configurados por el usuario y envía los datos a las salidas configuradas.
- **Auditbeat**: Sirve principalmente para auditar las actividades de los usuarios y los procesos en determinados sistemas. También puede usarse para detectar cambios en archivos críticos, como archivos binarios y de configuración, e identificar posibles violaciones de las políticas de seguridad entre otras cosas.
- **Heartbeat**: Es utilizado para comprobar periódicamente el estado de los servicios y determinar si están disponibles. Verificar que está cumpliendo con sus acuerdos de nivel de servicio para el tiempo de actividad y comprobar que nadie del exterior pueda acceder a los servicios en su servidor empresarial privado son algunos de los casos de uso más comunes.

Beats fue el último gran componente en añadirse al Elastic Stack para formar una pila más competente y con funcionalidades más específicas. Todas las piezas de Elastic interactúan entre ellas, tal como se ha visto a lo largo de este capítulo, y proporcionan un software realmente útil para desarrollar proyectos que requieran una recolección, almacenamiento y representación de datos. Es por esta razón, que para la implementación del presente proyecto, la cual se comentará en los siguientes capítulos, se ha hecho uso del conjunto de herramientas que componen el Elastic Stack.

Capítulo 4

Implementación

Una vez introducidas las herramientas y recursos que se utilizarán en el desarrollo del proyecto, así como el marco tecnológico en el que este tiene lugar, en este capítulo se procede a dar una descripción detallada de los pasos que se han seguido para realizar la implementación propiamente dicha. En este capítulo se mostrará una visión general de la arquitectura del sistema implementado, y se explicará cómo se han desarrollado y cómo interactúan cada uno de los elementos que la componen.

4.1. Arquitectura del sistema

Como ya se expuso en el capítulo introductorio, la implementación del sistema que concierne al presente proyecto tiene como objetivo final proporcionar una plataforma de visualización de la información de contexto relativa a *smart cities* de una manera sencilla.

La idea clave que subyace a este objetivo es conseguir desarrollar servicios que se puedan desplegar fácilmente en múltiples ciudades sin necesidad de realizar complejas configuraciones. Para posibilitar este comportamiento modular es necesario utilizar interfaces y modelos de datos estandarizados, es por ello que se va a hacer uso del API NGSI v2 y de los *Smart Data Models* para la implementación del proyecto.

A pesar de que la solución desarrollada podría ser replicada en otros entornos, para la realización de este proyecto se ha actuado solamente sobre la información de contexto generada

en la ciudad de Santander. De este modo se ha interactuado con datos reales generados en una *smart city* que sigue los estándares de API y modelos de datos necesarios para este caso de uso.

Para el desarrollo de estos servicios se ha hecho uso de los componentes proporcionados por el Elastic Stack (Elasticsearch, Logstash y Kibana), los cuales se integrarán con la plataforma de SmartSantander para permitir realizar múltiples visualizaciones de la información de contexto. El flujo desde que se generan los datos hasta que son accesibles por los usuarios pasa por diferentes componentes, cada uno de ellos con una funcionalidad específica.

La arquitectura del sistema completo puede dividirse en dos partes bien diferenciadas. Una de ellas es la parte de SmartSantander que ya está desplegada y que permite la generación de los datos y su adaptación a un estándar *de facto*. La otra es la parte que propiamente ha sido desarrollada en este proyecto, la cual recopila esos datos, los transforma y los almacena para después realizar representaciones gráficas.

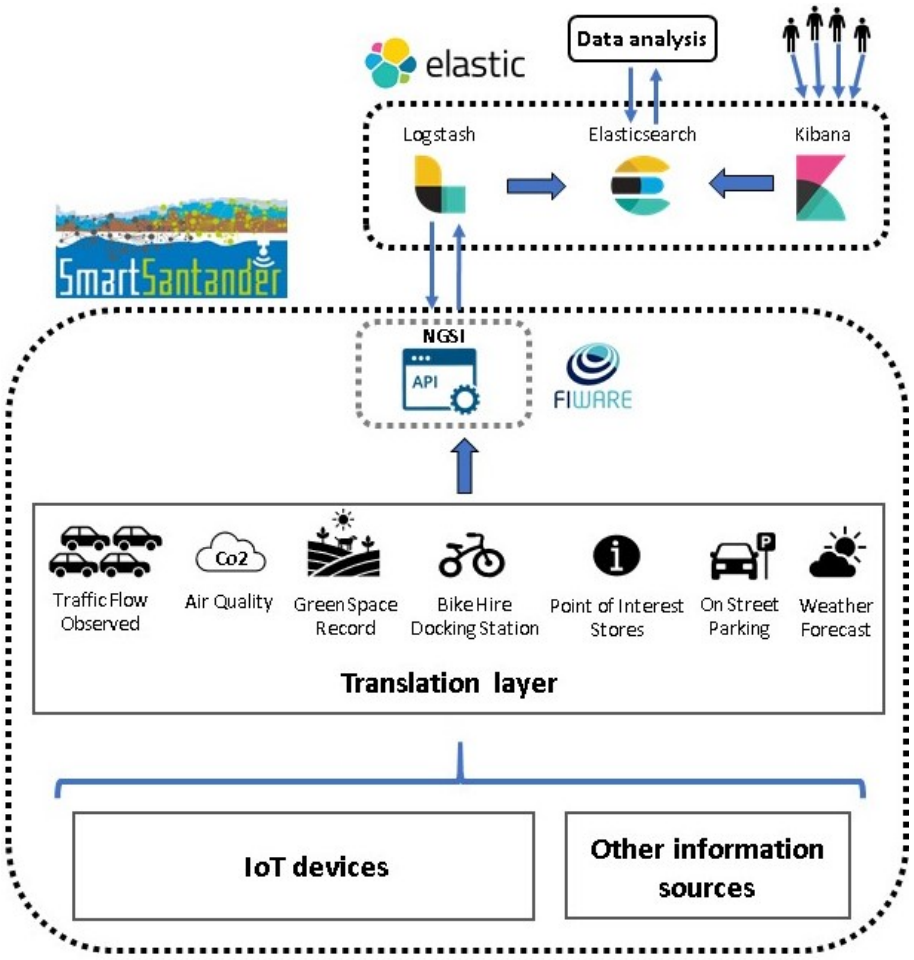

Figura 4.1: Arquitectura del sistema completo

La Figura 4.1 muestra una representación global del sistema en la que aparecen cada uno de los elementos que intervienen en él. Se puede apreciar la división del sistema en las dos partes antes comentadas, SmartSantander y Elastic. Para llegar a conocer cómo funciona cada uno de los componentes, a continuación se hará un análisis del flujo que siguen los datos desde su obtención hasta su representación.

El inicio del flujo de datos tiene lugar en la capa inferior de la arquitectura mostrada en la Figura 4.1 y viene marcado por la obtención de la información de contexto relativa a la ciudad de Santander. Una vez recopilados los datos, ya sea mediante dispositivos IoT u otras fuentes, se almacenan en un formato propietario en SmartSantander, y pueden ser consumidos mediante los diferentes APIs que expone la plataforma.

En concreto para la funcionalidad que se está desarrollando en este proyecto, se hará uso del API NGSI que expone SmartSantander. A su vez, la información expuesta por este API sigue las especificaciones definidas por los *Smart Data Models*. Para ello, previo a la publicación de los datos en el API NGSI, se ha de pasar por una capa de traducción (*Translation Layer* según la Figura 4.1) la cual tiene como objetivo convertir los datos con formato propietario de SmartSantander al estándar NGSI.

Existe una gran cantidad de sensores y fuentes de información accesibles mediante el interfaz NGSI, los cuales pueden agruparse en múltiples categorías (por ejemplo tráfico, puntos de interés, aparcamientos, etc) dependiendo de de la información que contengan. Cada una de esas categorías se correspondería con un *Smart Sata Model*, y por tanto se almacenaría en el API NGSI con su formato correspondiente.

Como la cantidad de categorías que se contemplan en las *smart cities* es bastante extensa, para la realización de este proyecto se ha actuado solamente sobre ocho *Smart Data Models*, los cuales han sido elegidos basándose en la disponibilidad de los datos, su calidad y su interés. A continuación se nombran esos ocho modelos de datos con los que se ha trabajado (también expuestos en la Figura 4.1):

- **`Traffic Flow Observed`**: Referente a datos del flujo de tráfico en la ciudad.
- **`Air Quality`**: Contiene datos de ciertas propiedades presentes en el aire.
- **`Green Space Record`**: Contiene datos de ciertos parámetros de las zonas verdes de la

ciudad.

- **`Bike Hire Docking Station`**: Referente a las estaciones para alquiler de bicicletas de la ciudad.
- **`Point of Interest`**: Aporta información acerca de los puntos de interés más importantes de la ciudad.
- **`Stores`**: Aporta información acerca de los comercios más importantes de la ciudad.
- **`On Street Parking`**: Referente a las plazas de aparcamiento en las calles.
- **`Weather Forecast`**: Aporta información acerca de la predicción meteorológica del día actual y del siguiente.

Una vez que la capa de traducción ha convertido los datos con formato propietario de SmartSantander a formato NGSI y los ha pasado por sus correspondientes *Smart Data Models*, los almacena en el API NGSI. Este almacenamiento sigue el formato NGSI comentado en anteriores capítulos, por lo tanto constará de entidades, atributos y metadatos (Figura 3.1).

Una vez que se almacenan los datos siguiendo los estándares en el API NGSI ya son accesibles desde el exterior de SmartSantander. Es aquí donde entra en juego la parte de la arquitectura del sistema que ha sido propiamente desarrollada durante la realización de este proyecto. Esta parte en la Figura 4.1 se referencia como Elastic, ya que se hace uso de los componentes proporcionado por esta compañía. En concreto se hacen uso de las tres herramientas más importantes del Elastic Stack: Logstash, Elasticsearch y Kibana.

Logstash es el encargado de hacer peticiones al API NGSI para obtener la información que se almacenó ahí, realizar transformaciones sobre ella y almacenar sólo los datos que se necesiten en Elasticsearch. Por otro lado, Kibana accederá a la información almacenada en Elasticsearch para realizar representaciones gráficas de los datos que serán accesibles por los usuarios finales de la aplicación.

Para completar el sistema, se ha generado un módulo cuya función principal es realizar análisis de datos para obtener más información de los datos existentes y así proveer un valor añadido a la aplicación. Este módulo realiza peticiones periódicas a Elasticsearch para obtener datos en un determinado periodo de tiempo, posteriormente aplica algoritmos de *machine*

learning para realizar predicciones acerca del valor de esos datos, y por último almacena esa predicción en Elasticsearch para que Kibana pueda representarla y los usuarios puedan hacer uso de ella.

Cuando se realizó el diseño de la arquitectura del sistema, en base a las necesidades que se tenían se decidió no hacer uso de la herramienta Beats del Elastic Stack. Esta decisión fue tomada teniendo en cuenta que Logstash es más eficiente a la hora de almacenar datos a gran escala y proporciona mayor flexibilidad para realizar transformaciones sobre ellos.

Una vez expuesto tanto el flujo de los datos como la arquitectura general del sistema completo, en las siguientes secciones se detallará el desarrollo explicando cada uno de los pasos tomados para llegar a la construcción del sistema final.

4.2. Obtención y almacenamiento de los datos

Tal como se ha comentado el anterior sección, los datos de contexto de la ciudad son accesibles mediante el API NGSI siguiendo el formato estándar de los *Smart Data Models*. Este trabajo de fin de máster tiene como objetivo final proveer un servicio que permita a los usuarios visualizar esos datos. Para que las prestaciones de este servicio sean mejores, los datos deberán estar almacenados en un lugar que permita acceder a ellos de manera rápida y eficaz.

Debido a que lo que se persigue es poder realizar búsquedas rápidas entre cantidades potencialmente grandes de datos, la mejor opción es utilizar bases de datos orientadas a documentos gracias a que su capacidad de indexación permite gran rapidez en las consultas. Es por esto que se ha tomado la decisión de utilizar Elasticsearch como almacén de datos de contexto, debido a que proporciona un motor de búsqueda realmente potente.

La elección de Elasticsearch como almacén de la información de contexto condiciona la decisión de qué herramienta utilizar para extraer los datos del API NGSI y ser inyectados en Elasticsearch. A pesar de que existen diferentes formas de realizar esto, la mejor integrada es Logstash.

Logstash tiene múltiples *plugins* que permiten extraer información de diferentes lugares, adaptarla mediante transformaciones y por último inyectarla en el destino que se desee,

Elasticsearch en este caso.

Para la inyección de datos, el primer paso es ejecutar Elasticsearch, que escucha en *localhost* en el puerto 9200. De este modo, se tendrá un *endpoint* al cual Logstash podrá enviar la información a almacenar. Antes de ejecutar Logstash, se comprueba que Elasticsearch está ejecutándose. En la Figura 4.2 se muestra la respuesta obtenida al hacer una petición para realizar dicha comprobación.

```
administrator@mario-nunez:~$ curl localhost:9200
{
  "name" : "mario-nunez.alumnos.tlmat.unican.es",
  "cluster_name" : "elasticsearch",
  "cluster_uuid" : "FQsgBA-jQEObl8sFExNRzw",
  "version" : {
    "number" : "7.9.0",
    "build_flavor" : "default",
    "build_type" : "tar",
    "build_hash" : "a479a2a7fce0389512d6a9361301708b92dff667",
    "build_date" : "2020-08-11T21:36:48.204330Z",
    "build_snapshot" : false,
    "lucene_version" : "8.6.0",
    "minimum_wire_compatibility_version" : "6.8.0",
    "minimum_index_compatibility_version" : "6.0.0-beta1"
  },
  "tagline" : "You Know, for Search"
}
```

Figura 4.2: Respuesta del *endpoint* donde se está ejecutando Elasticsearch

La ejecución de Logstash genera automáticamente métricas de tiempo de ejecución que pueden utilizarse para monitorizar su rendimiento. Esta herramienta proporciona APIs de monitorización para acceder a dichas métricas.

Para que Logstash comience a inyectar datos en Elasticsearch se ha de ejecutar un archivo de configuración de *pipeline*, el cual ya ha sido introducido en anteriores capítulos. En este archivo se van a definir todos los *plugins* de los que va a hacer uso Logstash para recoger información, transformarla e insertarla en el destino.

A pesar de que los *Smart Data Model* siguen un formato estándar que proporciona cierta uniformidad, al tratarse de datos de diversos contextos hará que en las respuestas del API NGSI aparezcan distintos campos dependiendo del tipo de datos que se esté pidiendo. Por ejemplo, los campos que definen datos de tráfico serán distintos a los que definen datos de la predicción

meteorológica. Es por esto que las transformaciones que vayan a aplicarse a esos datos serán también distintas. Por lo tanto, esto evidencia la necesidad de crear un archivo de configuración de *pipeline* de Logstash para cada modelo de datos.

A pesar de que se vaya a trabajar con ocho modelos de datos, y por tanto haya que crear ocho archivos de configuración de *pipeline* de Logstash distintos, la estructura de todos ellos es similar. Esto se traduce en que una vez generado el primero de estos archivos para un modelo de datos en concreto, la generación del resto solo requerirá de pequeñas modificaciones en el inicial.

Como se ha mencionado anteriormente, la estructura de los archivos de configuración de *pipeline* de Logstash se basa en la utilización de *plugins* que permitan obtener los datos del API NGSI, transformarlos de la manera deseada e insertarlos en Elasticsearch. Es por esto que dichos archivos de configuración estarán divididos en tres etapas bien diferenciadas: `input`, `filter` y `output`.

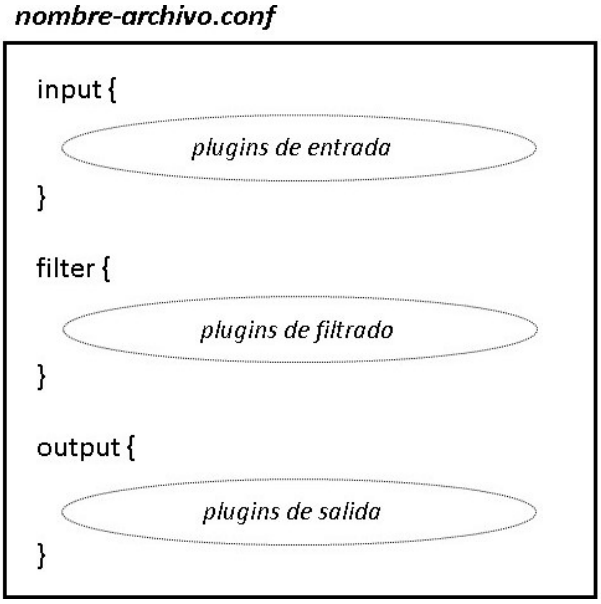

Figura 4.3: Estructura general de un archivo de configuración de *pipeline* de Logstash

En la Figura 4.3 se muestra un esquema general de un archivo de configuración de *pipeline* de Logstash, en el cual cada *plugin* utilizado deberá definirse en su etapa correspondiente para lograr realizar las acciones deseadas.

A continuación se hace una revisión de los *plugins* de entrada, filtros y salida utilizados en cada una de las etapas en los archivos de configuración de *pipeline* de Logstash generados para la realización del proyecto:

- **Etapa `input`**:

  Es la primera etapa del *pipeline* de Logstash y se define al inicio del archivo de configuración. Tiene como objetivo la extracción de datos y su conversión a eventos de Logstash. En nuestro caso, los datos están almacenados en el API NGSI, por lo tanto se realizarán peticiones HTTP para su obtención.

  Los datos de contexto medidos por los sensores cambian constantemente, por lo que para obtener su valor actualizado se deberán hacer peticiones periódicas al API NGSIo subscribirse al mismo. En concreto, se ha optado por las peticiones para tener un mayor control sobre la cantidad de datos almacenados. Es por esta razón que para la extracción de los datos se hace uso del *plugin* `HTPP Poller`. De todas las posibles configuraciones que soporta este *plugin*, se utilizan tres de ellas: *urls*, *request_timeout* y *schedule*.

  Lo primero que se ha de definir es la URL a la que se van a realizar las peticiones. Debido a que cada archivo de configuración de *pipeline* de Logstash deberá acceder a un tipo de dato (*Smart Data Model*), la URL definida en cada archivo deberá llevar asociados unos *query params* para filtrar por el tipo de entidad de la que se quiere obtener información. Además, se considera buenas prácticas limitar el número de resultados que se le piden al API a un millar.

  Por otro lado se define la periodicidad de las peticiones con el parámetro de configuración *schedule* y el tiempo de retransmisión mediante el parámetro *request_timeout*. Es importante que antes de definir la periodicidad de las peticiones, se tenga en cuenta el tipo de dato que se está accediendo. Por ejemplo, los datos de intensidad de tráfico están constantemente cambiando, pero los datos de predicción meteorológica solo se actualizan una vez al día.

  Por último, se aplica también el *plugin* llamado *json codec* para transformar los mensajes JSON que se extraen del API NGSI en eventos Logstash.

  A modo de ejemplo, en la Figura 4.4 se muestra un ejemplo de la obtención de los datos del *Smart Data Model* de `Traffic Flow Observed` mediante el archivo de configuración de Logstash.

```
input {
    http_poller{
        urls  => {
            test1  => http://10.10.38.10:1026/v2/entities?type=TrafficFlowObserved&limit=1000
        }
        request_timeout  => 20
        schedule  => { every  =>"120s"}
        codec => "json“
    }
}
```

Figura 4.4: Etapa de *input* de Logstash `TrafficFlowObserved`

Existen otro tipo de datos que son estáticos, como por ejemplo los de `Point of Interest` y `Store`. Este tipo de datos no llevan asociado una marca temporal, y por tanto no modifican su valor de manera automática con el paso del tiempo. Es por esto que en estos casos solo haría falta una única petición HTTP. Esto se podría conseguir utilizando el *plugin* `HTTP`, o tal como se ha realizado para mantener la misma estructura en todos los archivos, que es usando el *plugin* `HTTP Poller` y parando la ejecución después de que se realice la primera petición.

Es necesario considerar que en algunos escenarios el número de registros es superior al millar, como es el caso de los datos pertenecientes al tipo `Store`. El *plugin* `HTTP Poller` permite definir varias URLs a las que realizar la petición, por lo tanto la solución adoptada para este tipo de casos es realizar varias peticiones al mismo *endpoint* y en cada una de ellas pedir mil registros diferentes mediante la utilización de un *offset* definido en los *query params* de la petición. En la Figura 4.5 se puede observar la etapa de `input` generada para el *Smart Data Model* `Store`.

```
input {
    http_poller{
        urls  => {
            test1 => http://10.10.38.10:1026/v2/entities?type=Store&limit=1000&offset=0
            test2 => http://10.10.38.10:1026/v2/entities?type=Store&limit=1000&offset=1000
            test3 => http://10.10.38.10:1026/v2/entities?type=Store&limit=1000&offset=2000
        }
        request_timeout  => 20
        schedule  => { every  =>"1h"}
        codec => "json“
    }
}
```

Figura 4.5: Etapa *input* de Logstash múltiples peticiones

- **Etapa `filter`**:

  Una vez que se han extraído los datos del API NGSI como eventos de Logstash, es momento de procesarlos para adecuarlos a lo que se desea almacenar en Elasticsearch.

  A pesar de que los datos sigan el estándar de los *Smart Data Models*, es posible que dependiendo del servicio que se vaya a proveer se necesite realizar pequeñas transformaciones a estos modelos de datos. Todas estas modificaciones tienen lugar en esta etapa de filtrado.

  Existen dos motivos principales por los que se realizan estas transformaciones. El primero de ellos es eliminar todo tipo de información redundante o innecesaria presente en los datos, de este modo se consigue reducir su tamaño y por tanto se obtiene una mayor eficiencia a la hora de acceder a los mismos. Si se necesitase alguna información adicional relevante también se podría añadir. El segundo motivo es conseguir que cada uno de los campos que van a almacenarse en Elasticsearch lo haga con el tipo apropiado para que de este modo las representaciones gráficas que posteriormente van a hacerse con Kibana puedan visualizarse correctamente.

  Para poder aplicar estas transformaciones se hace uso del *plugin* de Logstash llamado `mutate`, el cual permite actuar de manera sencilla sobre los campos de eventos de Logstash. Las configuraciones que se han utilizado de este *plugin* son las siguientes: `add_field`, `replace`, `convert`, `gsub` y `remove_field`.

  Para ejemplificar visualmente el uso de este *plugin* y las configuraciones utilizadas, a continuación se muestra la transformación realizado al *Smart Data Model* `TrafficFlowObserved`. En la Figura 4.6 se representa de una manera visual la significativa reducción de los datos obtenida, en la cual los datos transformados que van a ser almacenados en Elasticsearch son de un tamaño significativamente menor a los obtenidos del API NGSI, lo cual evidencia claramente la utilidad de esta etapa de filtrado.

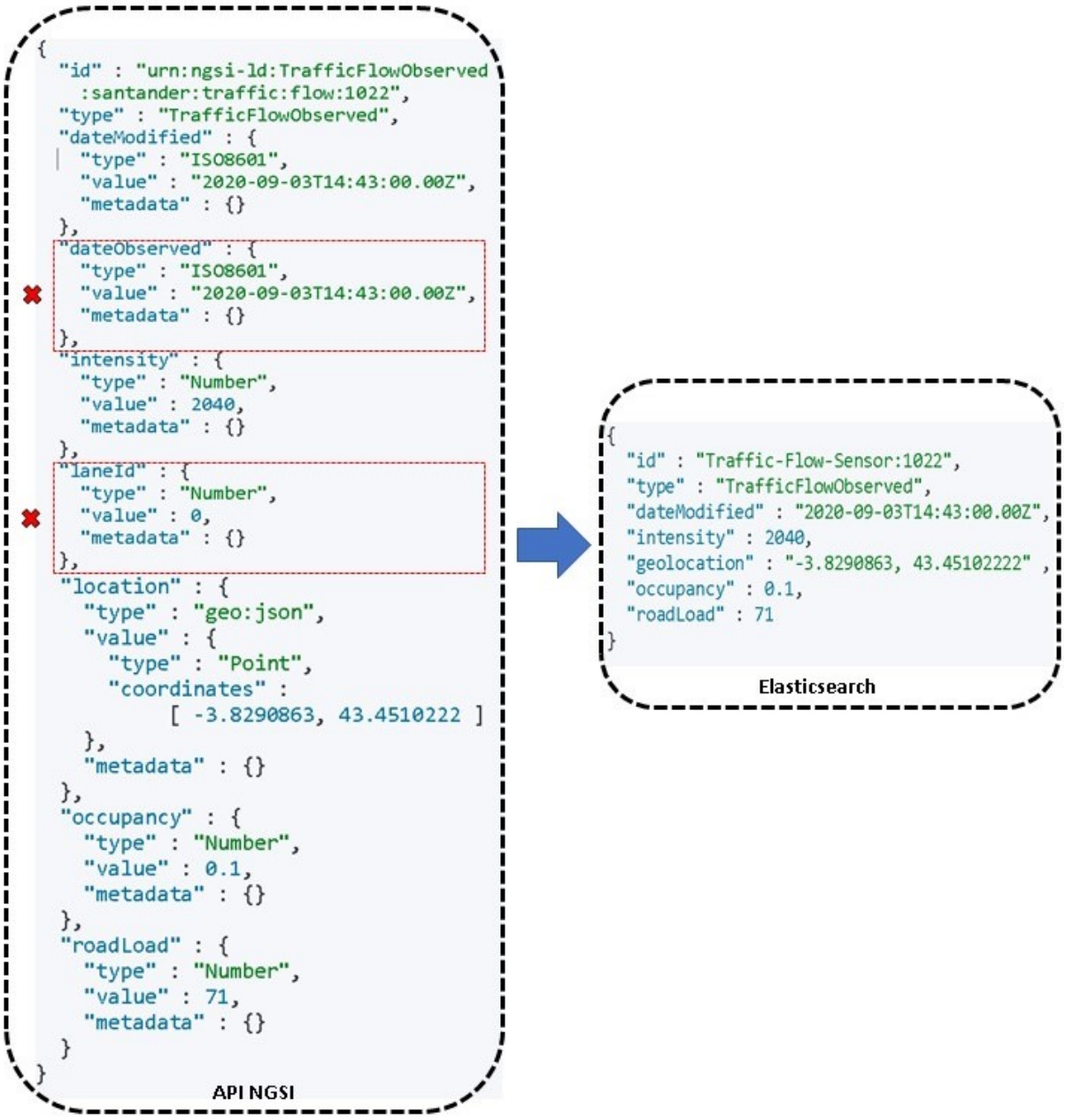

Figura 4.6: Transformación en etapa *filter* de `TrafficFlowObserved`

A continuación se detalla el procedimiento seguido para conseguir la transformación mostrada en la Figura 4.6, el cual sigue la misma filosofía en el resto de modelos de datos.

En primer lugar se han eliminado campos que, a pesar de formar parte del *Smart Data Model* estándar, no se van a utilizar. En este caso se trata de los campos `laneId` y `dateObserved`. Para llevar a cabo esta acción se ha hecho uso de la configuración `remove_field`.

Por otro lado, se han eliminado todas las propiedades de los atributos de cada campo a excepción del `value`. Esto es debido a que la estructura de los metadatos no está definida en los *Smart Data Models* y que el tipo es solo orientativo dentro de NGSI, por lo tanto no aportan una información relevante. Este es el caso de los campos `dateModified`, `intensity`, `occupancy` y `roadLoad`, de los cuales los tres últimos, al corresponderse con un número, se han convertido a formato *float* utilizando la configuración `convert` del *plugin*.

Además, se ha renombrado el identificador definido por los *Smart Data Models* para darle un nombre más corto y que sea más fácilmente comprensible por usuarios que vayan a hacer uso de él para obtener datos relativos a ciertas entidades. Esto ha sido posible gracias a la configuración `gsub`.

Por último, se ha actuado sobre el campo `location`, el cual representa la posición geográfica de la entidad. Este campo es especial ya que debe ser guardado en Elasticsearch de una manera específica para que posteriormente Kibana sea capaz de acceder a él y representarlo en un mapa. El procedimiento para tratar este campo consta de dos partes: la primera de ellas es conformar los datos tal y como se van a almacenar y la segunda es indicar a Elasticsearch que se está guardando un valor que representa una geolocalización.
La parte del tratamiento del campo de localización para almacenarlo en Elasticsearch se hace en la etapa `filter` del archivo de configuración de Logstash, y consiste en extraer los valores de latitud y longitud, convertirlos a tipo *float* y añadirlos al campo `geolocation`, tal como aparece en la Figura 4.6. Por otro lado, el indicar a Elasticsearch que el valor a guardar es una geolocalización se ha de realizar mediante el uso de plantillas de índice (*index template*), las cuales se explicarán más adelante en esta misma sección.

Una vez aplicadas todas las transformaciones anteriormente comentadas, se obtiene la misma información que aportaban los datos extraídos del API NGSI siguiendo los *Smart Data Models*, pero con una reducción significativa de su tamaño y una adecuación para facilitar en el futuro su representación con Kibana. Con esto se da por finalizada la etapa `filter` y se pasa a la última etapa del archivo de configuración de Logstash.

- **Etapa `output`**:

  Habiendo ya extraído los datos y aplicado las transformaciones oportunas sobre ellos, el siguiente y último paso es almacenarlos en uno o varios destinos para poder hacer uso de ellos en el futuro.

El principal objetivo de la etapa de `output` en este caso es almacenar los datos en Elasticsearch de manera ordenada y que permita realizar búsquedas rápidas para optimizar el acceso a los datos y mejorar así el funcionamiento de herramientas de representación como es el caso de Kibana.

Debido a que la base de datos en la que se van a almacenar los datos es Elasticsearch, para esta etapa se va a utilizar el *plugin* homónimo de Logstash, `Elasticsearch`.

Para almacenar los datos hace falta hacer uso de dos parámetros en la configuración del *plugin*, `hosts` e `index`. El primero de ellos sirve para indicar el destino dónde se van a almacenar, mientras que el segundo indica el índice bajo el cual se van a almacenar.

Como el destino es Elasticsearch, el cual se estaba ejecutando de manera local en el puerto 9200, el parámetro de configuración del `host` será `http://localhost:9200`.

Por otro lado, el parámetro `index` dependerá del tipo de dato que se esté guardando. Tal como se ha comentado antes, se está trabajando con diferentes *Smart Data Models*, cada uno de los cuales representa un ámbito de la ciudad sobre el que se puede obtener información (tráfico, tiempo meteorológico, aparcamiento, etc). Cada uno de estos ámbitos tendrá diferentes campos y por lo tanto deberá almacenarse en un índice distinto de Elasticsearch para luego poder realizar representaciones especializadas para cada uno de ellos. De este modo, cada archivo de configuración de *pipeline* de Logstash tendrá un `index` distinto dependiendo del tipo de dato que se esté almacenando.

En la Figura 4.7 se muestra la etapa `output` del *Smart Data Model* de los datos de tráfico (`TrafficFlowObserved`). En ella se ha definido un índice específico asociado a este *Smart Data Model* para su almacenamiento en Elasticsearch.

```
output {
    elasticsearch {
            hosts => http://localhost:9200
            index => "traffic“
    }
}
```

Figura 4.7: Etapa *output* de Logstash de `TrafficFlowObserved`

Cabe destacar que, aparte de los campos mapeados, también se va a almacenar en Elasticsearch el campo `@timestamp`, el cual es un campo de metadatos generado automáticamente por Logstash que indica la fecha en la que se ha procesado un evento. Su utilidad se explicará en detalle más adelante en el apartado de representación de los datos con Kibana.

Tal como se ha comentado anteriormente, los campos que representan coordenadas geográficas han de almacenarse en Elasticsearch de una manera concreta para que posteriormente puedan ser representados en un mapa. En la etapa de filtrado de Logstash se ha generado el campo `geolocation` compuesto por los valores de latitud y longitud, pero para que Elasticsearch entienda ese campo como unas coordenadas geográficas se le ha de indicar que el dato que está almacenando es del tipo *geo_point*. Debido a que Logstash no permite la conversión a tipo *geo_point*, se ha de actuar sobre la configuración de Elasticsearch para lograr esto.

La forma de indicar a Elasticsearch que el valor a guardar es una geolocalización se realiza mediante el uso de plantillas de índice (*index template*). Estas plantillas es una forma de decirle a Elasticsearch cómo configurar un índice cuando se crea, en este caso se le ha de indicar que cuando se almacene el campo `geolocation` deberá guardarse como tipo *geo_point*. En la Figura 4.8, se muestra la plantilla de índice que se ha utilizado para configurar los índices de Elasticsearch para que almacenen las geolocalizaciones con el tipo *geo_point*.

```
PUT _template/geotemplate
{
  "index_patterns" : [
      "traffic", "store", "bikestamp",
      "bike", "interest", "air", "green"
    ],
  "settings" : { },
  "mappings" : {
    "properties" : {
      "geoLocation" : { "type" : "geo_point" }
    }
  },
  "aliases" : { }
}
```

Figura 4.8: Petición para modificar el tipo del campo `geolocation`

En concreto la figura muestra una petición PUT realizada a Elasticsearch para indicarle que todos los índices definidos dentro de *index_patterns* llevarán asociado un campo `geolocation` el cual será de tipo *geo_point*. De este modo, las geolocalizaciones se almacenarán en Elasticsearch de la manera apropiada.

Si bien es cierto que los modelos de datos comparten similitudes, alguno de ellos también presenta ciertas diferencias con respecto al resto. De entre los ocho modelos de datos con lo que se trabaja en este proyecto, el de `OnStreetParking` es un tanto diferente en relación al almacenamiento de su geolocalización. Para todos los *Smart Data Models* que representen puntos en mapa se ha de realizar la configuración anteriormente comentada, en cambio en `OnStreetParking` se pretende representar rectas en el mapa las cuales van a estar asociadas con las calles de la ciudad en la que hay aparcamientos disponibles, esto hace que su configuración sea un tanto distinta.

Para almacenar datos de geolocalización en Elasticsearch como una recta se ha de hacer de un modo análogo a lo que se ha realizado con los puntos. Primero se tiene que utilizar Logstash para hacer un mapeo de la información y después se ha de actuar sobre las plantillas de índice para indicarle a Elasticsearch que se trata de información de geolocalización.

A modo de ejemplo, en la Figura 4.9 se muestra el mapeo que se ha realizado en la etapa de filtrado de Logstash del campo `location` del *Smart Data Model* `OnStreetParking` para almacenarlo en Elasticsearch.

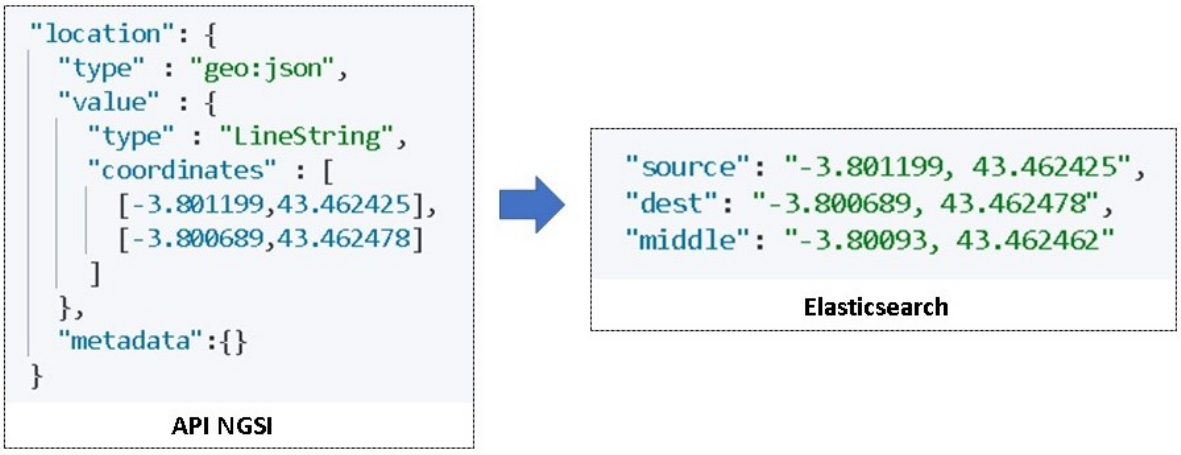

Figura 4.9: Mapeo de la localización en `OnStreetParking`

Tal como se observa, se ha convertido el campo `location` en tres campos diferentes. Los campos `source` y `dest` son extraídos del API NGSI y representan los puntos de origen y destino de la recta a representar. Además, se ha añadido el campo `middle` que contiene el punto geográfico medio de la recta. Dicho punto medio se utilizará a la hora de representar gráficamente los datos.

Por otro lado, se ha de utilizar una plantilla de índice para indicarle a Elasticsearch que estos campos que se han creado para representar una recta deberán ser de tipo *geo_point*. Para ello se utiliza la petición mostrada en la Figura 4.10.

```
PUT _template/geotemplateparking
{
  "index_patterns" : ["parking"],
  "settings" : { },
  "mappings" : {
    "properties" : {
      "source" : { "type" : "geo_point" },
      "dest" : { "type" : "geo_point" },
      "middle" : { "type" : "geo_point" }
    }
  },
  "aliases" : { }
}
```

Figura 4.10: Petición para modificar el tipo de los campos de geolocalizaciones en `OnStreetParking`

Tal como se puede observar, la plantilla de índice aplicada a Elasticsearch le indica que el índice `parking` va a llevar asociados los campos `source`, `dest` y `middle`, los cuales se almacenarán como tipo *geo_point*.

Una vez generados estos archivos, se ejecutan mediante Logstash de manera individual. Esto va a generar ocho flujos de datos distintos, cada uno de los cuales estará periódicamente insertando datos en Elasticsearch bajo el índice asociado al *Smart Data Model* al cual pertenezcan esos datos.

Los archivos de configuración de Logstash pueden ejecutarse desde la terminal de comandos. La instrucción que permite dicha ejecución sigue la estructura `bin/logstash [options]`. En el caso que concierne a este proyecto, las opciones utilizadas para ejecutar

cada uno de los ocho archivos de configuración han sido `-f` y `--path.data`. La primera de ellas lleva asociada la ruta al archivo de configuración y la segunda apunta a un directorio que permite escritura del cual Logstash hará uso cuando necesite almacenar ciertos datos.

4.3. Representación de los datos

En esta sección se detallará el procedimiento seguido para la generación de las representaciones gráficas de los datos que se mostrarán a los usuarios. Para cada uno de los tipos de datos de los *Smart Data Models* utilizados se ha generado un panel (*dashboard*) que ilustra los datos, para los se han utilizado diferentes tipos de representación. A lo largo de esta sección se explicarán los tipos de representación, acompañados por un ejemplo de uso.

Kibana actúa como una ventana a Elasticsearch, permitiendo de este modo visualizar todo lo que contiene dicha base de datos. Esta interacción se fundamenta en la indexación con la que se almacenan los datos en Elasticsearch, ya que Kibana permite realizar búsquedas y representaciones de dichos índices.

En el proyecto desarrollado, el haber actuado sobre ocho modelos de datos implica que se han creado a su vez ocho índices distintos en Elasticsearch, cada uno de los cuales está asociado a un modelo de datos concreto. El principal objetivo de Kibana es acceder a cada uno de esos índices para realizar las representaciones oportunas de dichos datos.

Para que Kibana pueda acceder a los datos asociados a un índice en Elasticsearch se ha de crear un *index pattern*, el cual permite tanto seleccionar los datos que serán utilizados como definir propiedades de sus campos. Como se tienen ocho índices distintos, se han de crear ocho *index patterns* para representar los datos contenidos en cada uno de ellos.

La creación de los *index patterns* se basa en dos pasos. El primero es seleccionar el índice de Elasticsearch al que están asociados y el segundo es seleccionar el campo con formato fecha sobre el que Kibana aplicará filtros temporales en el caso de que solo se quieran representar datos en un cierto periodo de tiempo.

Debido a que se están haciendo peticiones periódicas al API NGSI y obteniendo datos que a efectos prácticos se pueden considerar como en tiempo real, para la creación de los *index*

patterns se ha seleccionado el campo `@timestamp` que genera Logstash para que se apliquen sobre él dichos filtros temporales. En el caso de los *Smart Data Models* `PointOfInterest` y `Store`, se elige la opción de no utilizar un filtro de tiempo debido a que representan datos estáticos los cuales no varían con el tiempo.

Una vez que se han creado los *index patterns*, Kibana ya permite la creación de representaciones gráficas de cada uno de los campos asociados a esos índices. Tal como se ha comentado en la anterior sección, los tipos de datos que contienen esos campos son diferentes, algunos son numéricos, otros son fechas, otros son geolocalizaciones, etc. Es por ello que cada uno de ellos requerirá de una representación específica. A continuación se va a realizar una revisión de las visualizaciones utilizadas para representar diferentes tipos de campos.

Una de las visualizaciones que más se ha utilizado es la conocida como `Time Series Visual Builder (TSVB)`, la cual permite la representación de series temporales. Dicha visualización se ha utilizado para posibilitar la representación de la variación de campos con valor numérico a lo largo del tiempo.

Para la generación de visualizaciones con `TSVB`, se ha indicado el *index pattern* sobre el que se va a actuar y el campo que se quiere representar. Además, se ha seleccionado el campo `timestamp` asociado a cada medida para así poder formar la serie temporal. En la Figura 4.11 se muestra la serie temporal del campo `intensity` perteneciente al modelo de datos `TrafficFlowObserved` para un sensor en concreto en un intervalo de tiempo de dos horas.

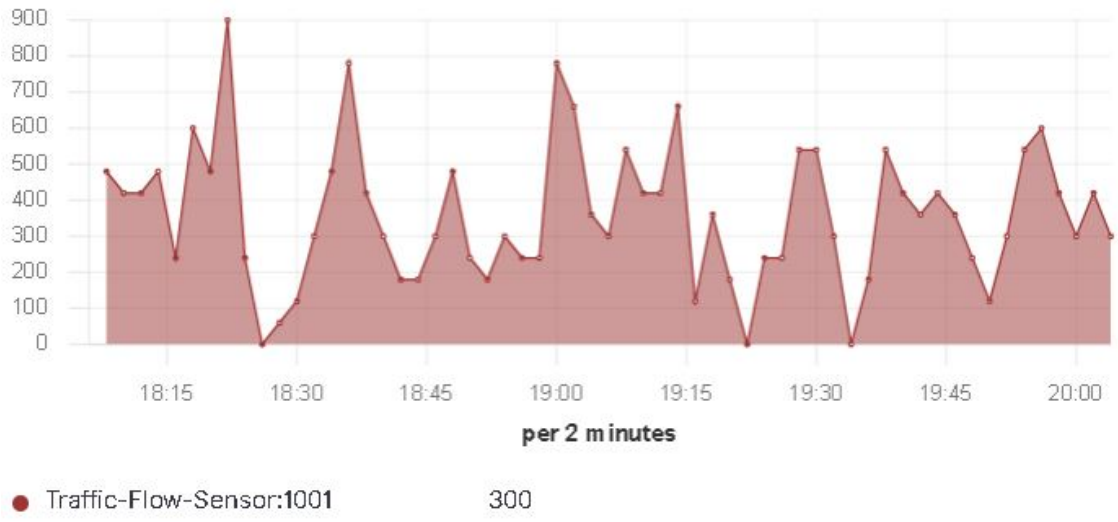

Figura 4.11: Serie temporal que representa la variación de la intensidad de tráfico

Tal como se puede observar en la imagen, el eje de ordenadas muestra la intensidad del tráfico mientras que el eje de abscisas representa la marca temporal (`@timestamp`). De este modo mediante el `TSVB` de Kibana se ha representado la variación de la intensidad de tráfico medida por un sensor en función del tiempo. Este tipo de visualización también permite representar varias medidas de diferentes sensores en la misma gráfica para comparar sus valores, tal como se mostrará más adelante en esta misma sección.

Cabe destacar que por norma general todos los campos de cualquier modelo de datos que hagan referencia a valores numéricos que varían con el tiempo pueden ajustarse a una serie temporal.

Además de representar series temporales, Kibana también permite generar gráficos de barras para mostrar el valor de un campo frente a otro, sobre los cuales se pueden aplicar diferentes operaciones como la media, la suma, valor máximo, valor mínimo, etc. En este proyecto, se ha creado este tipo de visualizaciones para los datos de tráfico debido a que este tipo de operaciones cobra más sentido sobre este tipo datos.

Para el modelo de datos `TrafficFlowObserved` se ha generado un histograma para representar el valor medio de sus campos numéricos (`intensity`, `occupancy` y `roadLoad`). A continuación, en la Figura 4.12 se muestran los valores medios de intensidad de varios sensores de tráfico en un intervalo de tiempo concreto.

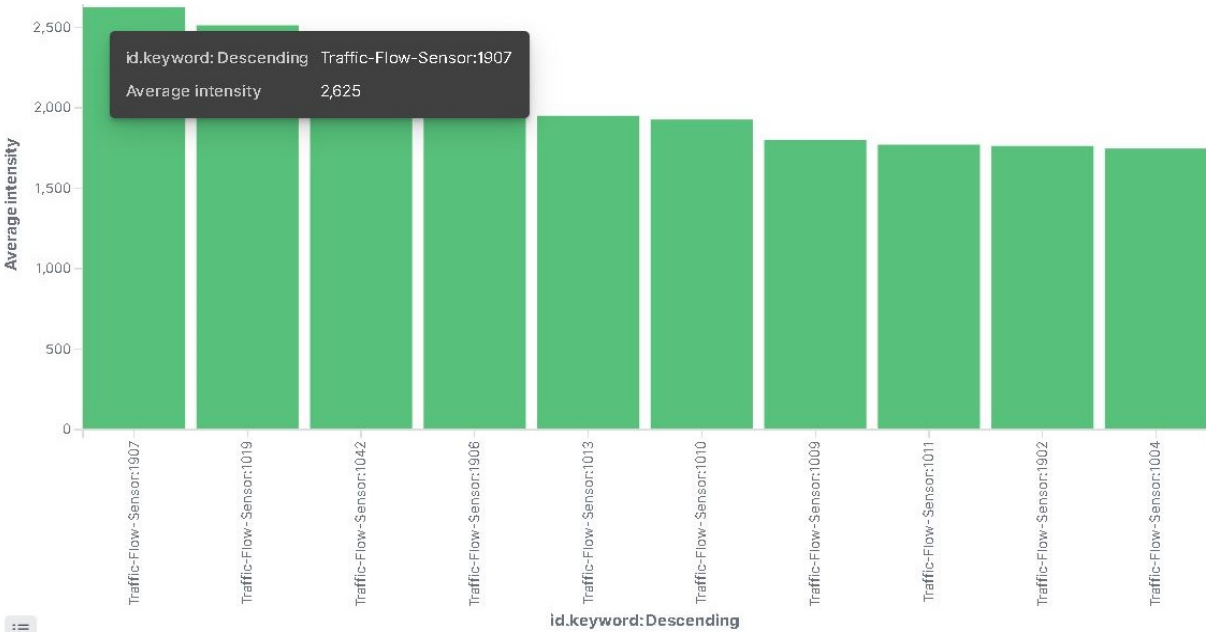

Figura 4.12: Histograma con los valores medios de intensidad de varios sensores de tráfico

Tal como se observa en la imagen, la visualización representa la media de los datos de intensidad de varios sensores en un cierto periodo de tiempo. Esto se muestra en forma de histograma en el que cada valor de intensidad se asocia con el identificador del sensor que ha tomado esas medidas.

Además de gráficos y series temporales, Kibana también permite la representación de coordenadas geográficas. Los modelos de datos con los que se está trabajando tienen campos con dos tipos de geolocalizaciones, puntos y rectas, los cuales se almacenaron previamente en Elasticsearch con el formato apropiado para permitir que la herramienta de Kibana los entienda como geolocalizaciones. Esto implica que dichas posiciones geográficas van a poder ser representadas mediante la herramienta `Maps` de Kibana.

Por un lado, en los *Smart Data Models* que poseen un campo de geolocalización representado por un punto se deduce que cada una de las medidas están siendo tomadas por un único sensor del que se conoce su posición geográfica. Es por eso que, por medio de este campo, en estos modelos de datos se han representado las localizaciones de los sensores en visualizaciones creadas mediante la herramienta `Maps`.

A continuación se muestra un ejemplo del mapa generado para el *Smart Data Model* `TrafficFlowObserved`, el cual contiene los datos relacionados con el tráfico de la ciudad. Los mapas generados para el resto de modelos de datos seguiría la misma filosofía que este.

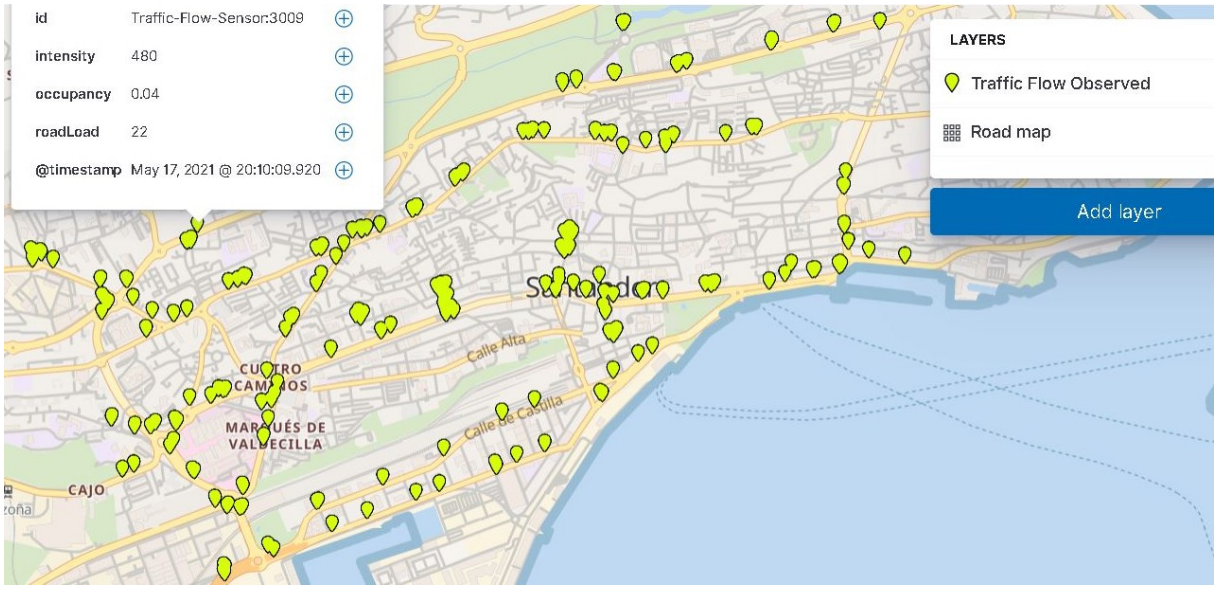

Figura 4.13: Mapa con las geolocalizaciones de los sensores de tráfico

Tal como se aprecia en la Figura 4.13, cada uno de los puntos del mapa representa la localización de un sensor de tráfico. La herramienta `Maps` permite además la definición de *tooltips*, por medio de los cuales al pulsar sobre alguno de esos puntos del mapa se abre una ventana con la información relativa a las diferentes medidas tomadas por ese sensor a lo largo del tiempo. Esto supone una gran ventaja para los usuarios ya que pueden conocer los últimos valores medidos por un determinado sensor tan solo pulsando un botón. En el *tooltip* que aparece desplegado en la Figura 4.13 se puede observar cómo aparecen los datos de un sensor de tráfico, ya que se está actuando sobre el modelo de `TrafficFlowObserved`.

En el caso concreto del *Smart Data Model* `PointOfInterest`, se representan datos estáticos acerca de los lugares de interés de la ciudad. Dicho modelo posee el campo `imageUrl`, el cual hace referencia a una URL que apunta a una imagen de dicho lugar. Para poder mostrar una imagen en el *tooltip* del mapa, se ha indicado desde Kibana que dicho campo tenga un formato de URL, de modo que al hacer cualquier representación de dicho campo se va a interpretar como tal. En la Figura 4.14, se muestra un ejemplo del *tooltip* en uno de los puntos del mapa generado para el *Smart Data Model* `PointOfInterest`.

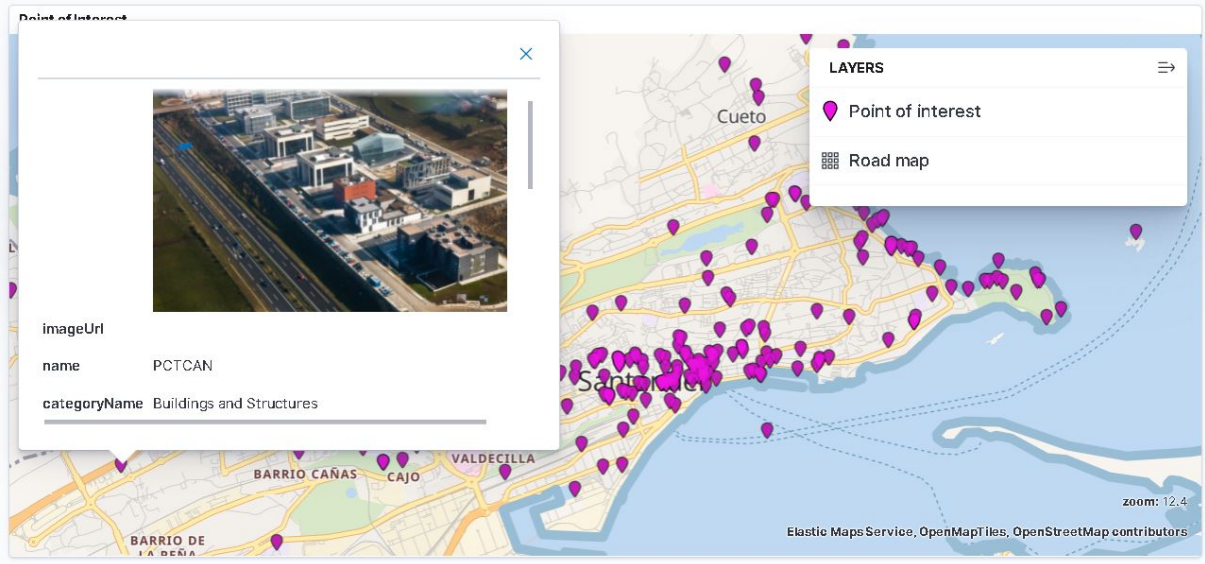

Figura 4.14: *Tooltip* con imagen en el mapa generado para `PointOfInterest`

Tal como se observa, no solo se ha añadido la imagen en el *tooltip*, sino que además se han incluido el resto de campos con información estática pertenecientes al modelo

`PointOfInterest`. De este modo, se consigue que cada uno de los puntos localizados en el mapa contenga toda la información referida a ese punto de interés de la ciudad.

Una vez abordadas las diferentes posibles configuraciones aplicables a puntos en el mapa, se procede a tratar el caso de los *Smart Data Models* que precisan de una representación geográfica con forma de recta. En este caso, para que Kibana entienda que se va a representar una recta, a la hora de crear el mapa se ha de indicar que va a ser del tipo *Point-to-Point*, así como los campos que van a actuar como inicio y fin de la recta.

De los *Smart Data Models* con los que se está trabajando, el único que posee una recta como geolocalización es el de `OnStreetParking`. Es por eso que para la creación de su correspondiente mapa en Kibana se seleccionó su campo `source` como origen y su campo `dest` como final de la recta.

Cabe destacar que la herramienta `Maps` de Kibana brinda la posibilidad de crear varias capas dentro de un mismo mapa. Se ha hecho uso de esta funcionalidad para añadir al mapa de `OnStreetParking` otra capa, aparte de la generada para representar la recta, con el punto medio de las rectas representadas, el cual está almacenado en el campo `middle`. En este punto medio de cada recta se ha añadido un *tooltip* para mostrar los datos de aparcamiento relativos a cada una ellas.

Por último, debido a que las rectas del mapa representan calles de la ciudad de Santander y poseen información acerca de los sitios de aparcamiento disponibles, se ha utilizado diferente color dependiendo del número de sitios disponibles que tengan.

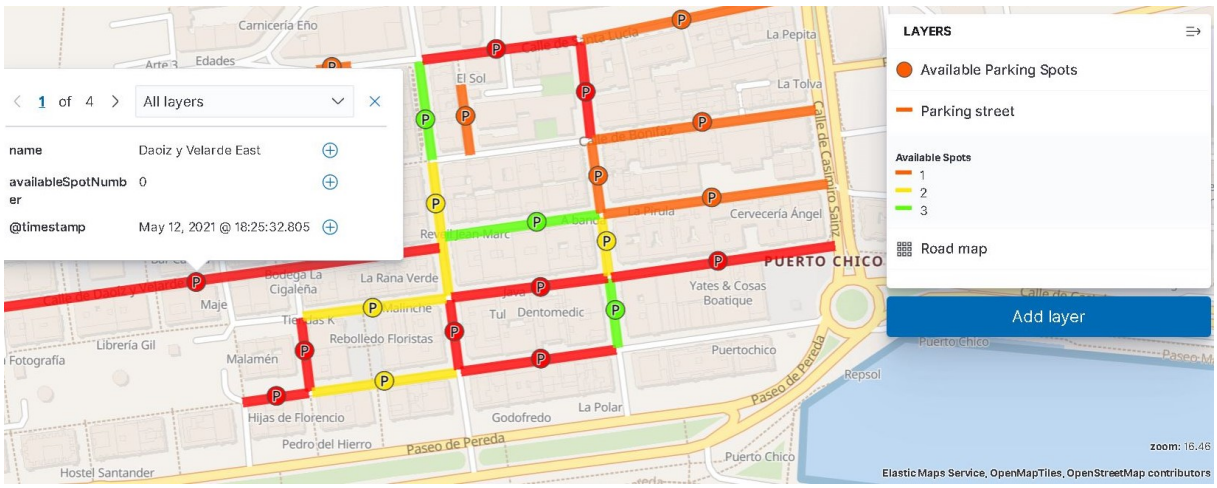

Figura 4.15: Mapa del *Smart Data Models* `OnStreetParking`

En la Figura 4.15 se puede apreciar cómo cada una de las rectas pintadas en el mapa representa una calle. Además, en el mapa aparece una leyenda en la que se especifica que en las calles mostradas con color verde hay más de tres sitios disponibles, en las de color amarillo hay dos, en las de color naranja hay tan solo uno y en las de color rojo todos los aparcamientos están ocupados. También se puede apreciar que en la imagen está desplegado el *tooltip* de una calle, y en él aparece tanto el nombre de dicha calle como los sitios que tiene disponibles en cierto momento.

Existen otros tipos de datos cuya representación no es tan visual ya que son simplemente datos de consulta. Este es el caso de las predicciones meteorológicas recogidas en el *Smart Data Model* `ForecastWeather`.

Los datos meteorológicos no tienen demasiado valor a lo largo del tiempo para los usuarios, ya que son simplemente predicciones que se realizan y que pierden su validez una vez que ha pasado el día al que se refieren. Por ello se ha considerado que la mejor forma de presentar estos datos es mediante una tabla informativa.

Para generar la visualización de la predicción meteorológica se ha hecho uso de la herramienta `Data Table` de Kibana, mediante la cual se han representado en forma de tabla los valores de los campos más representativos del modelo `ForecastWeather`. A continuación se muestra la visualización generada para este tipo de datos.

id	Day	Precipitation Prob.	Snow Level	Max Tª	Min Tª	Wind Speed	Wind Direction	Max Relative Humidity	Min Relative Humidity	Max UV index
ForecastWeather	May 17, 2021	0.25	1,700	16	12	5.6	-45	0.8	0.6	8
ActualWeather	May 16, 2021	0	0	17	13	0	-1	0.8	0.8	8
ForecastWeather	May 16, 2021	1	1,900	17	13	2.8	-90	1	0.75	8
ActualWeather	May 15, 2021	0	0	17	13	0	-1	1	0.9	8
ForecastWeather	May 15, 2021	1	0	18	13	1.4	45	0.95	0.75	8
ActualWeather	May 14, 2021	0	0	16	11	0	-1	0.8	0.7	7

Export: Raw Formatted

Figura 4.16: Tabla informativa de los datos del *Smart Data Model* `ForecastWeather`

Tal como se observa en la Figura 4.16, los datos meteorológicos son mostrados en una tabla informativa para facilitar la comprensión de los mismos por parte de los usuarios. Como se puede apreciar, para un mismo día aparecen dos datos, `ActualWeather` y `ForecastWeather`, los cuales son datos relativos a la predicción realizada el mismo día y el día anterior, respectivamente.

Una de las ventajas que tiene la representación en forma de tabla es que permite la descarga de los datos contenidos en ella en diferentes formatos. Esta funcionalidad puede resultar de utilidad para realizar análisis de datos y ver cómo de buenas han sido las predicciones meteorológicas realizadas.

Hasta ahora se han mostrado varios tipos de visualizaciones: serie temporal, diagrama de barras, mapa y tabla. En base a estos tipos de visualizaciones es posible generar las representaciones de todos los campos de los modelos de datos con los que se está trabajando. Para hacer esto simplemente se ha de conocer qué tipo de dato representa cada uno de los campos de los modelos de datos, y en base a esto representarle mediante la visualización que más se le ajuste. De una manera general se puede afirmar que las coordenadas serán representadas como un mapa, los valores informativos como una tabla y los valor numéricos que varíen con el tiempo como series temporales.

Una vez representados todos los campos de todos los modelos de datos se está en una situación en la que se han generado múltiples visualizaciones para cada uno de los distintos modelos de datos.

Para poder mostrar de manera conjunta varias visualizaciones pertenecientes a un mismo modelo de datos se generan paneles de información con Kibana, también llamados *dashboards*. Mediante esta herramienta se consigue agrupar la información y separarla atendiendo a su modelo de datos, consiguiendo así mostrar la información de una manera más clara.

La agrupación de la información en *dashboards* no solo aporta una mayor claridad para que los usuarios la consuman, sino que además brinda la posibilidad de añadir elementos con los que los usuarios puedan interacturar y filtrar los datos expuestos en las visualizaciones para obtener solamente la información que necesiten.

La herramienta de Kibana que se ha utilizado para permitir el filtrado de los datos en los *dashboards* es la conocida como `Controls`, la cual permite agregar entradas interactivas a los paneles de Kibana. La forma en la que se ha implementado en este proyecto es mediante el

uso de menús desplegables que permiten seleccionar una o varias entidades. De este modo los usuarios podrán obtener solamente la información de los sensores que deseen, sin necesidad de tener en las visualizaciones información que no quieren consultar acerca de otras entidades.

Por ejemplo, en el caso del *dashboard* del *Smart Data Model* `PointOfInterest`, se ha implementado un menú desplegable que actúa sobre el campo `category`. De este modo, se permite filtrar los resultados de los lugares de interés de la ciudad que van a aparecer en sus visualizaciones según la categoría a la que pertenecen esos lugares.

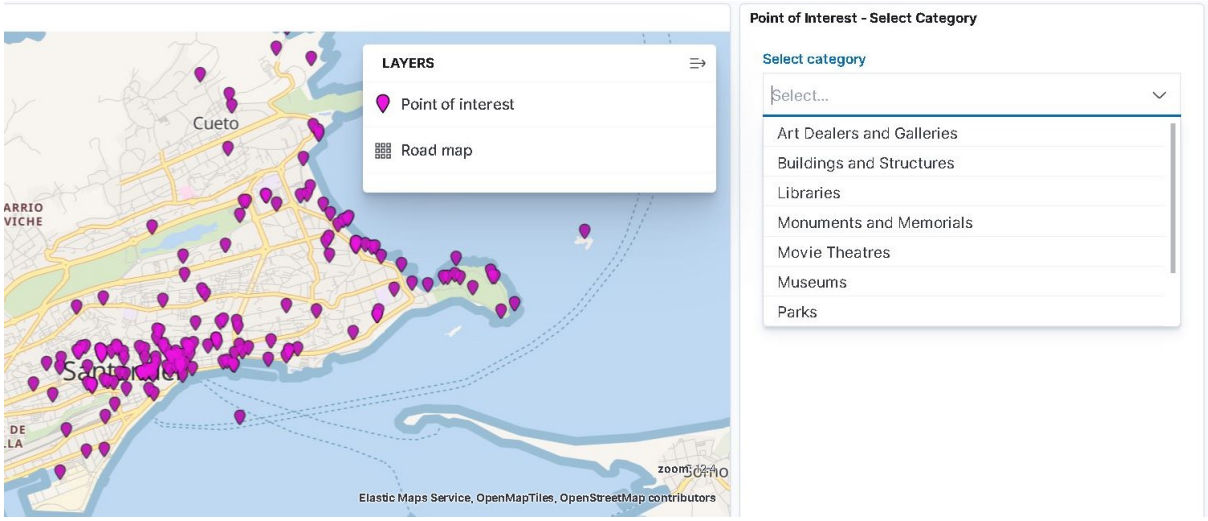

Figura 4.17: *Dashboard* del *Smart Data Model* `PointOfInterest` sin filtrar

Tal como se observa en la Figura 4.17, en el mapa aparecen todos los puntos de interés de la ciudad, los cuales pueden ser parques, bibliotecas, museos, etc, y mediante el menú de la izquierda se puede filtrar por el tipo de categoría que se desee visualizar.

Por otro lado, en algunos de los *dashboards* también se ha añadido una visualización de *Markdown*, en la cual se han incluido imágenes que actúan como enlace entre unos *dashboards* y otros. Esto resulta realmente útil en modelos de datos como `TrafficFlowObserved` en el que al haber tantos datos a representar no resultada adecuado mostrarlos en un solo panel.

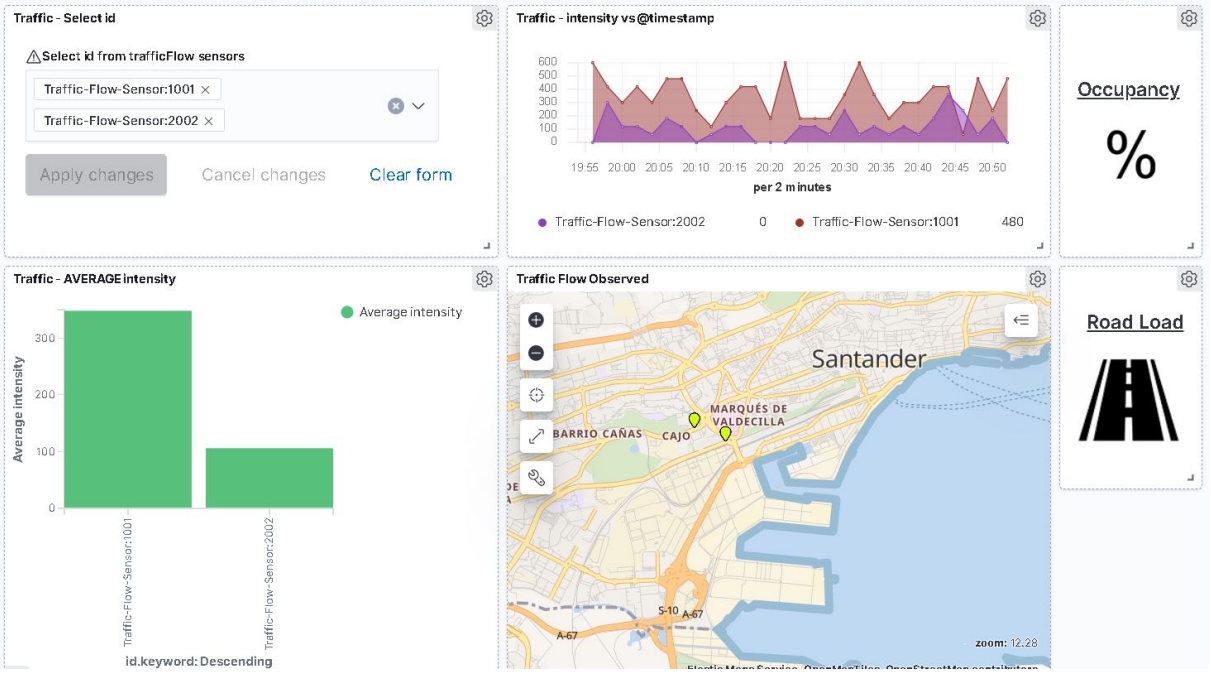

Figura 4.18: *Dashboard* de los valores de intensidad de tres sensores del *Smart Data Model* `TrafficFlowObserved`

En la Figura 4.18 se puede observar uno de los *dashboards* diseñados para el modelo `TrafficFlowObserved`. En él se aprecian las visualizaciones generadas para representar los datos de intensidad de tráfico. Se compone de un mapa con la localización de los sensores, una serie temporal con la evolución de la intensidad, un diagrama de barras con los valores medios de intensidad de cada sensor y un menú desplegable para filtrar por el campo identificador y así poder obtener solamente los datos de los sensores que se desea. A la derecha del *dashboard* se aprecian varias imágenes, las cuales actúan como enlaces, de modo que si se pincha sobre ellas redireccionan a otros *dashboards* de `TrafficFlowObserved` en los que se representan el resto de campos de este modelo de datos.
Concretamente en la Figura 4.18, se ha realizado ya el filtrado por dos sensores, `1001` y `2001`, es por eso que en cada visualización aparecen dos valores distintos. Gracias a esta interacción entre la herramienta de filtrado y el resto de visualizaciones del *dashboard* se pueden realizar comparaciones rápidas y visuales de los datos de diferentes sensores.

La unificación de las visualizaciones en diferentes *dashboards* permite a los usuarios tener agrupados en un mismo panel todos los datos relativos a un modelo de datos, lo cual resulta visualmente más agradable para ellos. Como el objetivo final es que un usuario pueda acceder a cualquiera de los *dashboards* generados, se ha creado también una página principal con enlaces

a cada uno de ellos. De esta forma todos los usuarios entrarían a la misma página y una vez dentro pueden seleccionar qué datos desean visualizar.

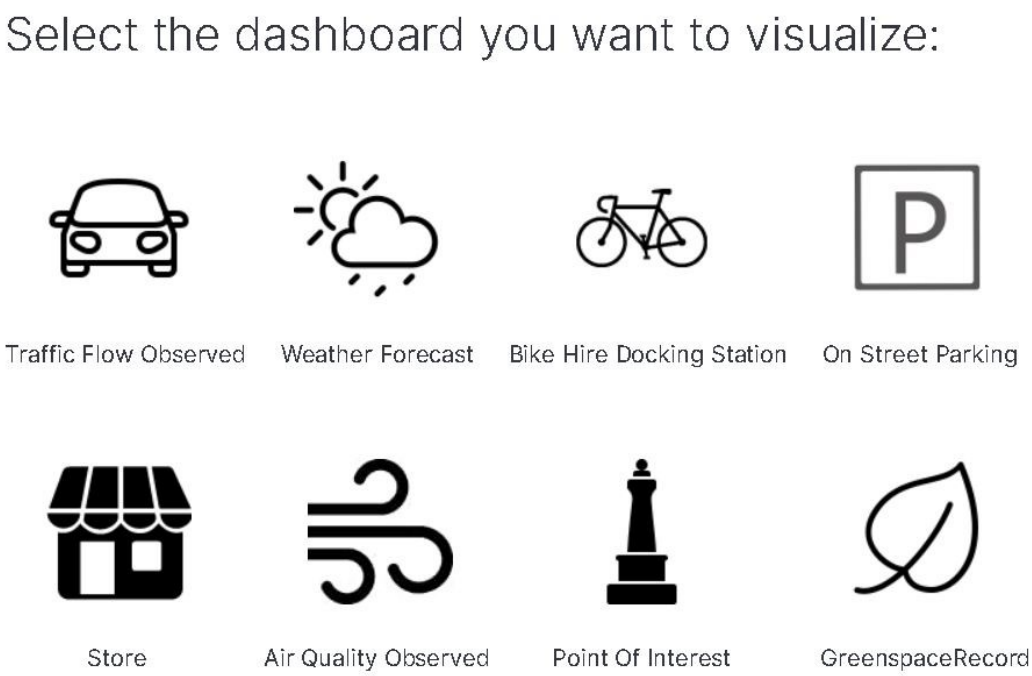

Figura 4.19: Página de inicio con enlaces a los *dashboards* generados

En la Figura 4.19 se muestra la página principal que se ha creado para que los usuarios puedan seleccionar el modelo de datos que quieren visualizar. Esta página está compuesta por elementos *Markdown* que contienen hipervínculos, en forma de imágenes y texto, que redireccionan a sus correspondientes *dashboards* cuando se pulsa sobre ellos.

Además de las funcionalidades ya comentadas, Kibana permite compartir los *dashboards* creados, lo cual puede hacerse mediante un enlace directo o incrustándolos en una página web. Esta característica dota de flexibilidad a la hora de desplegar el sistema desarrollado, permitiendo así adecuarlo a las diferentes situaciones de despliegue.

A modo de resumen, se ha accedido a los datos anteriormente almacenados en Elasticsearch y se han generado visualizaciones de los mismos con Kibana. Posteriormente se han agrupado esas visualizaciones por modelos de datos en distintos *dashboards* logrando así unificar la información para mostrarla de manera sencilla y permitiendo a los usuarios interactuar con

ciertos elementos de los *dashboards*. Además, se ha generado una página de inicio con enlaces a cada uno de los *dashboards* creados para facilitar la utilización de la aplicación.

Durante el desarrollo del presente proyecto tanto la transformación como la representación de los datos han sido realizadas para ocho modelos de datos diferentes, los cuales han servido como punto de partida para crear transformaciones y representaciones generales que permiten mostrar la funcionalidad del proyecto.

Gracias a la estructura consistente de los *Smart Data Models* es posible extender lo desarrollado a otros modelos de datos, es decir, de una manera similar y con cambios mínimos se pueden aplicar las mismas transformaciones y representaciones realizadas a otros modelos de datos.

Para conocer el grado de extensibilidad, se ha realizado un breve estudio del resto de *Smart Data Models* relativos a *Smart Cities* para determinar si lo que se ha desarrollado en este proyecto sería aplicable a ellos o no.

Antes de mostrar la tabla con los resultados del análisis de los *Smart Data Models*, se realiza una breve revisión de las columnas de la misma con el fin de clarificar los valores que pueden tomar cada una de ellas.

- **Dominio**: Se refiere al grupo donde se engloban varios *Smart Data Models*.
- **Smart Data Model**: Modelo de datos que se está analizando.
- **Transformación**: Hace referencia al nivel de modificación que requerirían los archivos de configuración de Logstash para adaptarse a determinado *Smart Data Model*.
  - **Uniforme (T_UNI)**: Solo requiere cambios mínimos en base a lo que se ha implementado. Dentro de esta categoría entrarían datos estáticos (no varían con el tiempo) y datos dinámicos (varían con el tiempo), así como los datos geográficos de puntos y rectas.
  - **Dedicado (T_DED)**: Requiere cambios mayores debido a que precisa de un tratamiento que aún no se ha implementado. Por ejemplo dentro de esta categoría entrarían los datos geográficos que representan superficies.

- **Representación**: Hace referencia a la forma en a que se muestran los datos visualmente.
  - **Estática (R_EST)**: Es un tipo de representación que solo requiere de cambios mínimos y que engloban datos que no dependen del tiempo y que geográficamente están representados por puntos o lineas.
  - **Dinámica (R_DIN)**: Es un tipo de representación que solo requiere de cambios mínimos y que engloban datos que varían con el tiempo y que geográficamente están representados por puntos o lineas. También se incluye en este apartado las representaciones de tablas dinámicas informativas como la del modelo `ForecastWeather`.
  - **Dedicada (R_DED)**: Hace referencia a una representación en la cual es preciso la creación de una visualización distinta debido a que las que se han realizado en este proyecto no se ajustarían correctamente a sus datos. En esta categoría entrarían modelos de datos con puntos geográficos móviles.
- **Aplicable**: Define si es posible representar un determinado *Smart Data Model* utilizando las funcionalidades implementadas en el proyecto.

Para realizar el estudio de los modelos de datos se contemplan todos los *Smart Data Models* relativos a *Smart Cities* a excepción de los pertenecientes al dominio `UrbanMobility` y que sigan el modelo GTFS de Google. Esto es debido a que los modelos de información GTFS y NGSI poseen filosofías muy diferentes y no aportaría demasiada utilidad.

En la Tabla 4.1 se ha denotado como transformaciones uniformes todas las desarrolladas durante este proyecto, así como las que fuesen muy similares a ellas. Las transformaciones y las representaciones dedicadas, `T_DED` y `R_DED` respectivamente, se refieren a modelos de datos que requerirían el desarrollo de una solución específica para poder hacer uso de ellos.

Tabla 4.1: Análisis de *Smart Data Models*

Dominio	Smart data model	Transformación	Representación	Aplicable
Building	Building	T_UNI	R_EST	✓
	BuildingOperation	T_UNI	R_DIN	✓
Parking	OffStreetParking	T_UNI	R_DIN	✓
	OnStreetParking	T_UNI	R_DIN	✓
	ParkingAccess	T_UNI	R_EST	✓
	ParkingGroup	T_DED	R_DED	✗
	ParkingSpot	T_UNI	R_DIN	✓
ParksAndGardens	FlowerBed	T_UNI	R_DIN	✓
	Garden	T_UNI	R_DIN	✓
	GreenspaceRecord	T_UNI	R_DIN	✓
PointOfInterest	Beach	T_UNI	R_EST	✓
	Museum	T_UNI	R_EST	✓
	PointOfInterest	T_UNI	R_EST	✓
	Store	T_UNI	R_EST	✓
Ports	BoatAuthorized	T_UNI	R_DED	✗
	BoatPlacesAvailable	T_UNI	R_DIN	✓
	BoatPlacesPricing	T_UNI	R_DED	✗
	SeaportFacilities	T_UNI	R_DED	✗
Streetlighting	Streetlight	T_UNI	R_DIN	✓
	StreetlightControlCabinet	T_UNI	R_DED	✗
	StreetlightGroup	T_DED	R_DED	✗
	StreetlightModel	T_UNI	R_DED	✗
Transportation	BikeHireDockingStation	T_UNI	R_DIN	✓
	CrowdFlowObserved	T_UNI	R_DIN	✓
	EVChargingStation	T_UNI	R_DIN	✓
	Road	T_UNI	R_EST	✓
	RoadSegment	T_UNI	R_EST	✓
	TrafficFlowObserved	T_UNI	R_DIN	✓
	TransportStation	T_DED	R_DED	✗
	Vehicle	T_UNI	R_DED	✗
	VehicleModel	T_UNI	R_DED	✗
UrbanMobility	ArrivalEstimation	T_UNI	R_DED	✗
	PublicTransportRoute	T_UNI	R_DED	✗
	PublicTransportStop	T_UNI	R_DED	✗
WasteManagement	WasteContainer	T_UNI	R_DIN	✓
	WasteContainerIsle	T_UNI	R_EST	✓
	WasteContainerModel	T_UNI	R_DED	✗
Weather	WeatherAlert	T_UNI	R_DIN	✓
	WeatherForecast	T_UNI	R_DIN	✓
	WeatherObserved	T_UNI	R_DIN	✓

Los resultados del estudio de los *Smart Data Models* quedan recogidos en la Tabla 4.1, en ella se puede apreciar cómo algunos de ellos aparecen como no aplicables. En la mayoría de los casos se debe a que los modelos de datos no se ajustan a ninguna de las representaciones que se han generado para los ocho modelos de datos con los que se ha trabajado. Por otro lado, al resto de ellos se les podría aplicar las transformaciones y representaciones ya generadas sin necesidad de grandes cambios.

Este estudio proporciona información de utilidad en el caso de que en un futuro se decida extender las funcionalidades para representar todos los *Smart Data Models* relativos al dominio de las *Smart Cities* para así proveer a los usuarios un servicio más completo. Para concluir se hace un balance positivo del estudio, ya que de todos los modelos de datos analizados, a la mayoría de ellos se les podrían aplicar las funcionalidades desarrolladas en este proyecto.

4.4. Técnicas de analítica de datos

En esta última sección del capítulo se va a explicar en detalle el módulo de analítica de datos implementado en la arquitectura global del sistema. Se van a tratar aspectos como el procedimiento seguido para su desarrollo, la interacción que tiene con el resto de componentes del sistema y la gran variedad de posibilidades que brinda para realizar análisis de datos.

La correcta utilización de los datos disponibles conlleva grandes beneficios, como pueden ser desde una mejora en la toma de decisiones por parte de las organizaciones, hasta una mejor provisión de servicios para los clientes. La mejor forma de conseguir estos beneficios es analizando los datos existentes para extraer aún más información de la ya existente.

El análisis de datos puede definirse como un proceso de inspección, limpieza, transformación y modelado de datos con el objetivo de obtener información útil que pueda aportar algún tipo de beneficio para la entidad que esté haciendo uso de estas técnicas. Existen muchas formas en las que los datos pueden ser utilizados para obtener un beneficio de ellos, pero por norma general la analítica de datos suele agruparse en cuatro tipos bien diferenciados [13]:

- **Análisis descriptivo**: Este es el tipo de análisis de datos más básico y común de todos. Se basa en analizar los datos obtenidos con anterioridad, para así poder brindar información valiosa sobre lo que ha sucedido en el pasado. Contesta a la pregunta *¿Qué ha pasado?*.
- **Análisis de diagnóstico**: Es el siguiente nivel de complejidad en el análisis. Trata de encontrar la causa por la cual se han obtenido ciertos valores en el pasado. Contesta a la pregunta *¿Por qué ha pasado?*.
- **Análisis predictivo**: Este tipo de análisis utiliza datos almacenados previamente para predicciones lógicas acerca de eventos futuros. Contesta a la pregunta *¿Qué es probable que suceda en el futuro?*.
- **Análisis prescriptivo**: Se basa en combinar la información extraída de los tres primeros tipos de análisis anteriormente mencionados para determinar el curso de acción a seguir en un problema o decisión actual.

El tipo de análisis más apropiado dependerá del entorno en que se aplique y del objetivo que se quiera conseguir. En el proyecto al que se refiere este documento se ha decidido generar un módulo de análisis predictivo.

Las predicciones que van a realizarse se harán mediante la utilización de algoritmos de *machine learning*, los cuales pueden aplicarse desde diferentes partes, incluso desde dentro del Elastic Stack. Tal como se ha comentado en anteriores capítulos, Kibana dispone de funcionalidades de *machine learning* para actuar sobre los datos, pero debido a que estas funcionalidades son de pago se decidió desarrollar soluciones específicas para el propósito que se quiere obtener.

Por esta razón se ha decidido implementar un módulo independiente capaz de extraer datos de Elasticsearch, aplicar algoritmos sobre ellos y generar predicciones que serán insertadas de nuevo en Elasticsearch. Cabe destacar que el propósito de la implementación no es el desarrollo de soluciones de predicción precisas, si no el de proporcionar la interacción con el resto de la plataforma. En este sentido, la implementación se ha validado con una solución de predicción sencilla, dejando como trabajo futuro el desarrollo de soluciones de predicción más precisas que hagan uso del módulo desarrollado.

El módulo de análisis de datos se ha generado en el lenguaje Python. Esta decisión fue tomada debido a dos razones principales. La primera es que dicho lenguaje es uno de los más

utilizados en analítica de datos, más concretamente en técnicas de *machine learning*. La segunda razón por la que se decidió utilizar Python es debido a que Elasticsearch proporciona un cliente oficial para este lenguaje el cual permite una fácil interacción y posibilita la extracción e inserción de datos de manera sencilla.

La predicción de datos puede hacerse mediante diferentes técnicas, es por eso que se va a realizar una revisión de las prestaciones aportadas por diferentes algoritmos para después poder elegir con criterio el más apropiado para el proyecto que se está desarrollando. Los cinco algoritmos que van a analizarse son los siguiente:

- ***Simple Random Forest***: Genera múltiples de árboles de decisión y los fusiona para obtener una predicción más precisa y estable. Se puede utilizar para tareas de clasificación y regresión [14].
- ***Grid Random Forest***: Trata de mejorar la actuación del algoritmo *Random Forest* mediante el ajuste de alguno de sus parámetros como por ejemplo el número de árboles de decisión.
- ***Neural Network***: Las redes neuronales tratan de simular el comportamiento del cerebro humano para reconocer patrones y tomar decisiones en base a ellos [15].
- ***Linear Regression***: Los modelos de regresión lineal estudian la relación entre una variable dependiente y una o más variables independientes. Su uso más típico es para el análisis predictivo [16].
- ***Support Vector Regression (SVR)***: Es un modelo que se fundamente en algoritmos SVM aplicados a problemas de regresión. SVR aporta flexibilidad y permite definir cuánto error es aceptable en un modelo para encontrar una línea apropiada que se ajuste a los datos [17].

El módulo de análisis de datos ha sido diseñado para que extraiga ciertos valores de Elasticsearch, realice una predicción, genere métricas acerca de la precisión de dicha predicción y gráficas que comparen los valores predichos con los valores reales medidos. Las métricas y las gráficas generadas por el módulo son una herramienta para comparar el desempeño de los algoritmos aplicados.

Para realizar la predicción se han definido dos parámetros. El primero de ellos es el intervalo de tiempo en el que se van a extraer valores ya almacenados en Elasticsearch sobre los cuales se va actuar. El segundo es la longitud del histórico que va a utilizarse, es decir, cuántos valores anteriores van a usarse para realizar la predicción.

Para realizar la comparación de las prestaciones de los diferentes algoritmos se ha decidido extraer los valores de intensidad de tráfico del modelo `TrafficFlowObserved` de una semana, debido a que se precisa de una gran cantidad de datos para que el análisis sea relativamente fiable. Además, para hacer más liviana la ejecución de los algoritmos se han agrupado dichos valores extraídos por horas y se ha realizado la media. De este modo, la media de la intensidad de tráfico en cada hora actúa como un valor histórico. Por último se ha fijado que para realizar la predicción se utilice un número configurable de valores previos. A continuación se realiza la comparación de las diferentes gráficas obtenidas en las ejecuciones de los algoritmos.

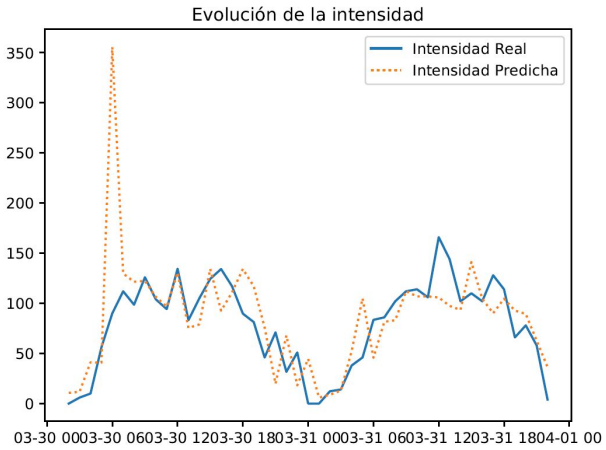

Figura 4.20: *Random Forest*

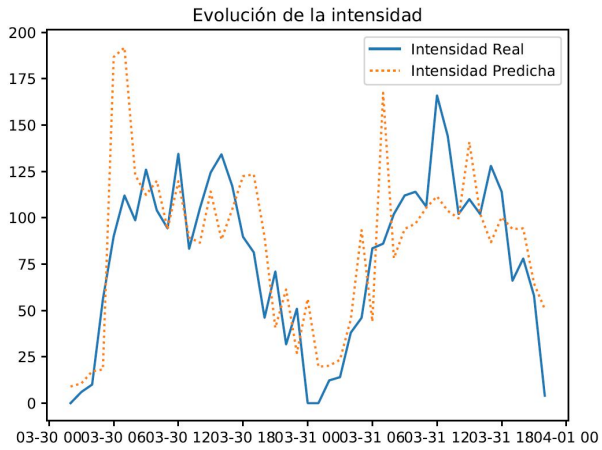

Figura 4.21: *Grid Random Forest*

En las Figuras 4.20 y 4.21 se puede apreciar cómo se ajustan los valores predichos a los reales según los algoritmos *Random Forest* y *Grid Random Forest*, respectivamente. Se puede ver que la predicción sigue más o menos la tendencia de los valores reales, pero hay ciertos puntos en los que se aleja significativamente. Observando las gráficas se deduce que la actuación de *Grid Random Forest* es un tanto mejor que la del *Random Forest* debido a que realiza el ajuste de los parámetros del algoritmo para obtener mejores resultados.

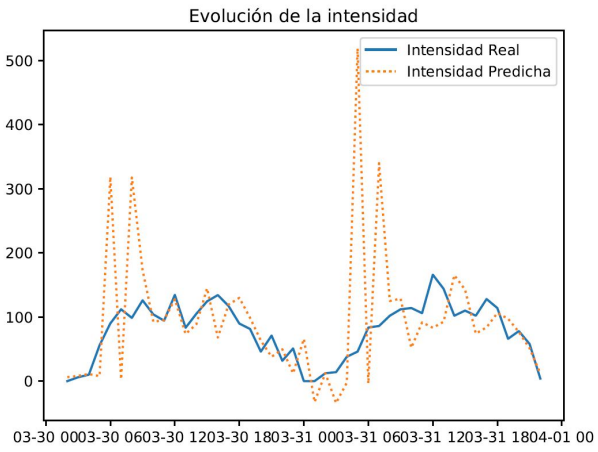

Figura 4.22: *Neural Network*

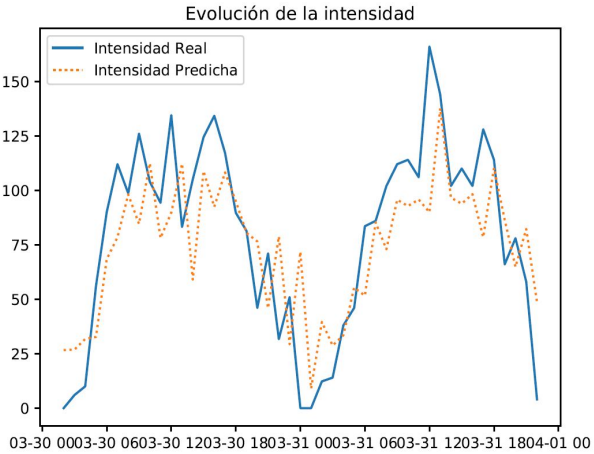

Figura 4.23: *Linear Regression*

Por otro lado, en las Figura 4.22 y 4.23 se representa la actuación de los algoritmos *Neural Network* y *Linear Regression*, respectivamente. En primer lugar se puede apreciar que la predicción obtenida por la red neuronal no es demasiado buena ya que los valores predichos difieren bastante de los reales. En cambio, mediante el algoritmo de regresión lineal se obtienen predicciones que tienen un cierto parecido a los valores reales. La gráfica obtenida mediante el algoritmo *Support Vector Regression (SVR)* no se ha incluido debido a que era prácticamente igual a la de la regresión lineal.

A la vista de todas estas comparaciones, se ha decido implementar el algoritmo de regresión lineal. Esta decisión se basa en dos motivos, el primero de ellos es que se obtiene la predicción que más se acerca al valor real para este caso de uso, y el segundo es porque su ejecución es una de las más liviana debido a la sencillez del algoritmo.

Una vez que se ha decidido que se va a utilizar un algoritmo de regresión lineal que hace uso de dos valores de históricos se va a definir el funcionamiento del módulo de analítica de datos desarrollado. Básicamente este componente consiste en un *script* de Python que hace uso de diversas librerías para realizar tres funciones generales: extracción de datos de Elasticsearch, entrenamiento del modelo de regresión lineal con los datos extraídos para realizar predicciones e inserción en Elasticsearch de los resultados obtenidos.

Como el objetivo principal del proyecto no es este módulo de análisis de datos, sino que es simplemente un servicio de valor añadido, lo que se busca es definir la estructura del sistema e implementar una sencilla funcionalidad de predicción. Es por este motivo que solamente se

ha actuado sobre el modelo de datos `TrafficFlowObserved` para realizar predicciones de los valores medios de intensidad de tráfico que tendrá en la siguiente hora un determinado. De este modo, si en un futuro se decidiese extender este componente al resto de sensores y tipos de datos ya se tendría una estructura consistente sobre la que simplemente haría falta aplicar los algoritmos de análisis de datos oportunos.

De cualquier manera, el beneficio que subyace a la realización de la predicción de intensidad media en la siguiente hora se percibe mejor cuando se quiere conocer información acerca de una agrupación de sensores en una zona geográfica de la ciudad, por ejemplo un municipio o un conjunto de calles. Mediante el conocimiento de la intensidad de tráfico media de un *cluster* de sensores, se pueden realizar predicciones acerca de la intensidad media que se tendrá en esa zona concreta de la ciudad, lo cual aportaría una idea general acerca del tráfico que habrá en dicho lugar.

Una vez que se tienen los valores medios de intensidad de tráfico, se entrena el modelo con ellos, lo que permite usarlo para predecir valores futuros. Esta predicción obtenida ha de insertarse en Elasticsearch para poder hacer representaciones con Kibana. El resultado del algoritmo que ha de guardarse es una predicción, por lo que no se considera un dato puramente del modelo `TrafficFlowObserved`, es por esto que se ha decidido almacenar los valores medios predichos de intensidad de tráfico bajo un nuevo índice en Elasticsearch, llamado `prediction-intensity`.

Para poder realizar representaciones de los datos de intensidad media de tráfico predichos y poder compararlos con los reales, se ha decidido insertar en Elasticsearch junto a la predicción otros campos complementarios, los cuales son:

- **`predictionAverageIntensityFuture`**: Este campo hace referencia al valor predicho de intensidad media de tráfico para la siguiente hora.
- **`predictionDate`**: Esta fecha es para la cual se ha realizado la predicción.
- **`realAverageIntensityPresent`**: Este campo hace referencia al valor de intensidad media de tráfico en la hora actual.
- **`realDate`**: Esta fecha es en la cual se tiene el valor actual de intensidad.

- **`sensorId`**: Hace referencia al identificador del sensor sobre el que se están realizando predicciones.

Una vez que se ha puesto en funcionamiento el sistema de analítica de datos que introduce valores en Elasticsearch se generan las visualizaciones oportunas en Kibana para que los usuarios puedan acceder a estos datos de predicción del mismo modo que para los diferentes modelos de datos.

Debido a la naturaleza de los datos que se están insertando en Elasticsearch, se ha decidido que la mejor forma de mostrarlos es mediante series temporales. Para ello se hace uso de la herramienta `TSVB` de Kibana, con la cual se generan dos gráficas, una con los valores medios de intensidades de tráfico reales y otra con las predicciones. Dichas series temporales usarán como campo de tiempo `realDate` y `predictionDate`, respectivamente. Tal como se ha realizado en anteriores representaciones, las visualizaciones generadas se agrupan en un *dashboard* para facilitar su interpretación.

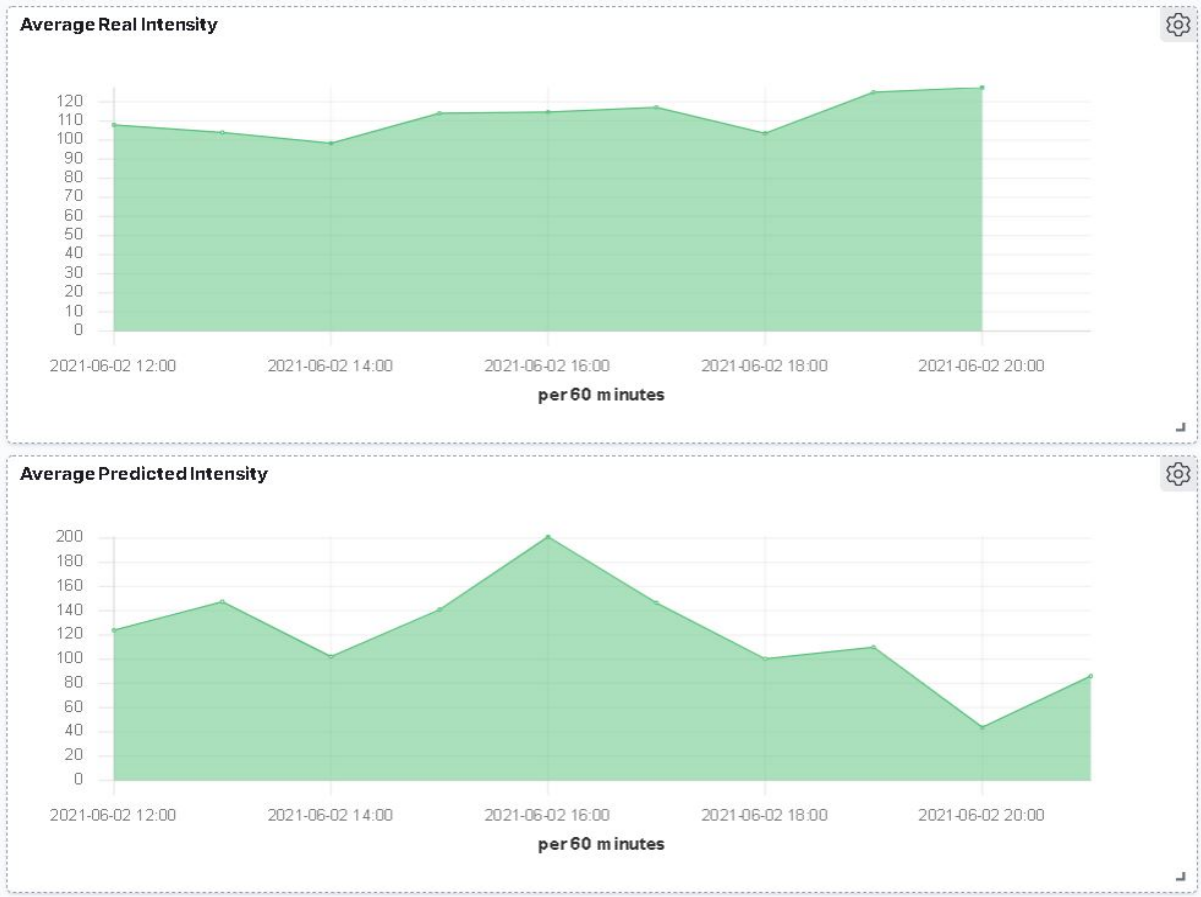

Figura 4.24: *Dashboard* con series temporales de intensidad de tráfico media real y predicha

En la Figura 4.24 se puede observar el valor medio real de la intensidad de tráfico del sensor `2040`, mientras que en la gráfica inferior se está representado el valor predicho. Según los valores temporales del eje de abscisas, el *dashboard* muestra la comparación entre los valores reales y los predichos desde las 12 de la mañana hasta las 9 de la noche del día 2 de junio de 2021. Tal como se aprecia, a las 21:00 aparece un valor predicho de intensidad, pero no uno real, esto es debido a que a las 20:00 se ha ejecutado el *script* de Python que realiza la predicción de la siguiente hora, pero esa hora aún no posee un valor real de intensidad.

A la vista de la comparación de los valores de intensidad medios reales y predichos de la Figura 4.24, se puede determinar que la predicción no es buena, permite validar la interacción entre el módulo de analítica de datos y el resto de los componentes.

Capítulo 5

Conclusiones y líneas futuras

Este capítulo final se reserva para la exposición de las conclusiones obtenidas al finalizar el desarrollo del proyecto. Además, se proponen una serie de posibles ampliaciones del mismo con el fin de abrir varias lineas de desarrollo que podrán ser abordadas en el futuro a partir de este trabajo.

5.1. Conclusiones

La posibilidad de extraer información de contexto en ámbitos como las ciudades permite obtener grandes cantidades de datos que pueden ser usadas para proveer servicios a los ciudadanos. Por norma general, cada localidad posee unos determinados dispositivos IoT para realizar esta tarea, lo que supone que el formato de los datos sea diferente según el fabricante o la plataforma utilizada. Este escenario supone que para desplegar un mismo servicio en dos localidades que manejan tipos de datos diferentes habría que realizar numerosos ajustes, lo que supondría un coste elevado.

Una de las posibles soluciones a esta situación es tratar de aportar uniformidad y consistencia a los datos. Para dicho propósito lo más común es hacer uso de modelos de información y de datos estándares, de modo que cualquier servicio generado para una localidad puede ser fácilmente extensible a otra, siempre y cuando compartan estos estándares.

En base a esto se percibe la necesidad de estandarizar y de hacer servicios homogéneos

cuyo despliegue en otras localidades sea sencillo y automático. En respuesta a esa necesidad, en este trabajo se ha implementado un sistema de representación y visualización de datos que siguen el estándar NGSI y la iniciativa *Smart Data Model*. En concreto, la solución desarrollada se ha basado en el entorno Elastic Stack, que es ampliamente utilizado por la comunidad de desarrolladores, y se han usado datos de SmartSantander.

El proceso de adaptación de Elastic Stack a las necesidades y tipos de datos presentes en SmartSantander se ha basado tanto en la realización de transformaciones uniformes de los modelos de datos como en la generación de representaciones gráficas que, al presentar también cierto grado de uniformidad, permiten replicar su funcionalidad de manera sencilla en cualquier *smart city* que proporcione datos que sigan los estándares utilizados.

Además, se ha implementado también un módulo de análisis de datos que interacciona con Elastic Stack y que permitiría generar información de valor añadido.

El resultado final del proyecto desarrollado son una serie de servicios homogéneos que, a pesar de haber sido desarrollados para una *smart city* en concreto, podrán ser utilizados tanto por ciudades como cualquier entidad que obtenga datos consistentes y estandarizados de la misma forma. De este modo se incentiva a las organizaciones a utilizar estándares para modelar sus datos, ya que así podrían simplemente utilizar servicios como los que se han desarrollado en este proyecto sin necesidad de tener que generarlos desde cero.

5.2. Líneas futuras

El desarrollo de este Trabajo de Fin de Máster no solo proporciona un sistema para proveer servicios homogéneos a los usuarios, sino que además ha sentado las bases para futuras ampliaciones y mejorar de lo hasta ahora desarrollado.

En primer lugar, gracias al estudio que se ha realizado de los modelos de datos estándares pertenecientes a las *smart sities*, los conocidos como *Smart Data Models*, se ha podido proporcionar una tabla en la que se recogen cada uno de los modelos de datos a los que podrían extenderse los servicios generados. Por lo tanto, la primera ampliación posible de este proyecto sería la de desplegar estos servicios para todos los modelos de datos a los que se puedan adaptar.

Aparte de tratar todos los modelos de datos posibles, también se podrían desplegar los servicios generados en otras localidades que compartan la misma estandarización en modelos de información y de datos Santander, la cual es la ciudad en la cual se ha trabajado para el desarrollo de este proyecto.

Otra de las posibles líneas futuras que puede seguir este trabajo es lograr adaptar lo desarrollado al interfaz estándar NGSI-LD. Esta adaptación supondrá también adaptarse a modificaciones en los modelos de datos de acuerdo a esta interfaz.

Por último, gracias a la estructura de analítica de datos generada, se tiene la posibilidad de realizar algoritmos más complejos y profundos para tratar de sacar el máximo partido a los datos existentes aportando de esta forma más servicios de valor añadido a los ya existentes.

Bibliografía

[1] Tai hoon Kim, Carlos Ramos and Sabah Mohammed. Smart City and IoT. *Future Gener. Comput. Syst.*

[2] Bengt Ahlgren, Markus Hidell and Edith C.-H. Ngai. Internet of Things for Smart Cities: Interoperability and Open Data. *IEEE Internet Comput., vol. 20, no. 6, pp. 52–56, Nov. 2016*.

[3] Kevin Ashton. That 'Internet of Things' Thing. *RFID journal*, 2009.

[4] John Kosowatz. Top 10 Growing Smart Cities. *ASME*, 2020.

[5] Smartsantander official web. www.smartsantander.eu. [Última consulta 25/04/2021].

[6] Fiware official web. www.fiware.org. [Última consulta 10/03/2021].

[7] FIWARE Open API Specifications. API NGSI v2. www.fiware.github.io/specifications/ngsiv2/stable. [Última consulta 15/03/2021].

[8] Smart data models official web. www.smartdatamodels.org. [Última consulta 02/04/2021].

[9] Elastic official web. www.elastic.co. [Última consulta 14/04/2021].

[10] James Turnbull. *Elasticsearch: The Definitive Guide*. 2016.

[11] Gurpreet S. Sachdeva. *Practical ELK Stack*. 2017.

[12] Clinton Gormley and Zachary Tong. *The Logstash Book*. 2015.

[13] Patrick Gibson. Types of Data Analysis. *Chartio*.

[14] Niklas Donges. A Complete Guide to the Random Forest Algorithm. *Built In*.

[15] IBM Cloud Education. Neural Networks. 2017.

[16] Shrikant I Bangdiwala. Regression: simple linear. 2018.

[17] Tom Sharp. An Introduction to Support Vector Regression (SVR). 2020.